# A UNIPROCESSOR IMPLEMENTATION OF FP FUNCTIONAL LANGUAGE

Leon Alkalaj

April 1986
CSD-860064

UNIVERSITY OF CALIFORNIA
Los Angeles

# A Uniprocessor Implementation Of FP Functional Language

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Leon Alkalaj

1986

To my parents

# Contents

# List of Illustrations

# Acknowledgments

I would like to thank my advisors prof. Milos Ercegovac and prof. Tomas Lang for their guidance and their insistence on quality. Additionally I would like to thank Luis Monsalve Jr. for his insightful comments and Miquel Huguet, Dan Greening and Dorab Patel for their fruitful discussions.



† This work has been supported in part by ONR Contract N00024-83-K-0493.

# Acknowledgments

I would like to thank my advisors prof. Milos Ercegovac and prof. Tomas Lang for their guidance and their insistence on quality. Additionally I would like to thank Luis Monsalve Jr. for his insightful comments and Miquel Huguet, Dan Greening and Dorab Patel for their fruitful discussions.



† This work has been supported in part by ONR Contract N00024-83-K-0493.

## ABSTRACT OF THE THESIS

A Uniprocessor Implementation Of FP Functional Language

by
Leon Alkalaj
Master of Science in Computer Science
University of California, Los Angeles, 1986
Professor Milos Ercegovac, Co-Chair
Professor Tomas Lang, Co-Chair

A uniprocessor implementation of the FP functional language is considered. The model of computation and implementation is described and the implementation constraints are discussed. A pointer data structure for the implementation of FP objects in memory is chosen.

Three memory management policies: Sequential, Linked List and a Stack implementation are considered and compared for memory allocation and garbage collection performance. Matrix Multiplication and Quicksort are benchmarks used to compare performance results.

The Stack approach to memory management is dynamic and therefore avoids the overflow overhead present in both the Sequential and Linked List approach. The implementation overhead in the Stack policy is also significantly smaller. Besides this, it is simpler, more flexible and leads to a high locality of memory references. It is also more amenable for a multiprogramming implementation and avoids many of the drawbacks of the static policies considered. One of the main drawbacks of the Stack memory management policy is the high cost of cell allocation which includes dynamic garbage collection operations.

A method for storing the garbage data structures on the stack is proposed in order to keep the staek size small. A hardware stack implementation in the form of a register file and hardware support for fast cell allocation is suggested for a VLSI on chip implementation.

With hardware support and a register file for garbage collection the overhead of memory management, typically encountered in uniprocessor implementations of functional languages, is significantly reduced.

# 1. Introduction

Functional Languages represent an alternative approach to the programming practice found in Imperative Languages. The term Functional Languages refers to languages that are based on function application and are free of side effects. The basic operation, therefore, in functional programming is function application. The value of a function is determined only by its arguments. Examples of functional languages are FP [Backus78], KRC [Turner82], Val [McGr79], ID [Arvind78] and others [Vegd84].

Claims have been made that functional languages increase the programmer's productivity, efficiency, compactness of coding and ease of program verification. Backus [Backus78], for example, advocates the use of functional languages as a means of reducing the memory-processor bottleneck, that is, the "Von Neumann Bottleneck". Others [Morr80] on the other hand, question whether functional programming is suitable for "real applications such as text editors, operating systems or video games", [Vegd84].

One of the main properties of functional languages is called Referential Transparency [Kelle81], [Turner81]. This means that there is no program or time dependency imposed on functions or objects. Once a function is defined, regardless of where or when in the program it is invoked, it will always perform the same operation, given the same input object.

Functional programming is free of side effects. This means that a programmer can construct a program without considering the possibility of aliasing occurring, or even the possibility that other routines might interfere. The only way a function may affect another function is through its output object.

Debugging programs written in a functional language may be easier than debugging programs written in imperative languages [Morr80]. That is, one could trace down the execution tree and examine for each function the input and output objects. This seems easy enough to do since for a given input object, the execution tree is static. Debugging an imperative language would consist of examining the state of computation.

Since side effects are inherent to the imperative programming style, one would have to follow the history of changes of the state machine.

Functional languages offer a variety of constructs that contain easily detectable and implicitly defined parallelism. This makes them attractive for possible multiprocessor implementations.

Even though the use of the functional programming style has been advocated by its supporters, functional programming has its drawbacks. For example, programming an inherently sequential algorithm, like I/O operations, in a functional language is difficult since they represent side effects. A possible answer to this problem is offered by Landin [Land65]. He suggests using Streams, that is, a representation of list structures implemented by passing the elements sequentially. It is not yet clear whether this solution is general enough to implement sequential algorithms using the functional programming style.

Other drawbacks attributed to the functional programming approach are related to their inefficient implementations. Problems that account for the lack of speed include memory management overhead, especially noticeable during garbage collection, and the overhead due to the high frequency of function calls and parameter passing. Another issue to be noted is that most of the implementations of functional languages have so far been interpreted rather than compiled. This would also account for a significant reduction in the speed of execution.

## 1.1. Research Goal And Motivation

In the past few years, and especially since Backus's Turing lecture [Backus78],there has been a growing interest in functional languages as an alternative to the conventional programming style of Imperative languages. The recent technological breakthroughs in VLSI have lead many computer architects, especially within university centers, to implement special purpose processors for the support of a particular language. Even though, up to the present, a special purpose VLSI processor for the execution of functional languages has not been implemented, most implementations confirm existing inefficiency problems.

The main goal of the research effort described within this report is to offer an implementation model for the functional language FP, and to address issues in the implementation of functional languages that relate to the problems encountered in memory management.

Most previous implementations of FP address alternative parallel architectures to exploit the implicit concurrency offered by the functional programming style and various models of interpretation. The main issues considered here are related to the problems encountered in memory management on a single processing element. The algorithms developed are applicable to both uniprocessor and multiprocessor implementations.

The initial interest in this research area originated from a class project report within the seminar on High Speed Computing, CS 259, held by prof. M.D.Ercegovac [Alkal84], [Monsal84]. The basic ideas offered in these reports were further developed, implemented and are thus presented.

## 1.2. Previous Research

A number of architectures have been designed for the execution of functional languages. Most recently Huynh, Hoevel and Hailpern proposed a new execution architecture based on Johnston's contour model [Johnst71] and on Hoevel and Flynn's notion of DEL/DCA architectures [Flynn83]. They refer to their architecture as DELfp, that is, a Directly Executed Language for FP.

Castan and Organick suggest a HLL RISC processor architecture for the parallel execution of FP language programs [Castan83]. They extend their model to include the broad class of Lisp Like Languages (3L-form). The processor architecture they proposed was designed with VLSI considerations.

Other less recent, but nevertheless significant, implementations include Patel's [Patel80] multiprocessor reduction machine which consists of a number of identical processing elements arranged in a ring structure interconnected by queues. Similarly, Treleaven and Mole [TrMo80] suggest a ring machine architecture consisting of identical execution units interconnected by double-ended queues. Mago [Mago80] proposes a tree network of processors, Xiong [Xiong84] a machine execution based on queues and Kellman [Kellm83] a parallel reduction machine with a row of processing elements connected through a common sorting network.

An excellent summary of some of the proposed architectures is presented in [Vegd84].

## 1.3. Thesis Organization

Even though it is presumed that the reader is familiar with the functional language paradigm, a short summary was presented within this introductory chapter. The main goals, motivations and previous research in this area are also summarized.

In Chapter 2 the functional language FP is described. A simple example illustrating an FP program is presented. Four FP programs, later used as benchmarks, are introduced here and are fully given in Appendix A.

In Chapter 3 the Uniprocessor Implementation of FP is described by presenting the model of computation and execution. Interpretation versus compilation issues are discussed and implementation constraints are recognized.

Chapter 4 contains an analysis of the two data structures that were considered for the implementation of FP. The two are compared in terms of speed of execution and implementation overhead. Further implementation, simulation and performance measurements are made only for the chosen data structure.

Chapter 5 contains the description of the Memory Management Simulator that was implemented using an already existing interpreter for FP. The simulator support routines referenced here are listed in Appendix B. Three different memory management policies are then introduced and implementation constraints are discussed. A detailed analysis of the performance parameters are presented for each implementation. The performance estimates are evaluated first for a general implementation and then for a specific host processor. The three implementations are finally compared using benchmarks.

In the Discussion section of Chapter 5, a memory management policy for FP is proposed based on the results previously described.

In the concluding Chapter 6 the work thus presented is summarized. Future directions and research goals are also discussed.

# 2. The FP Functional Language

In this chapter, a brief overview of the FP functional language is described. It is presumed that the reader is familiar with the functional programming style. Examples of FP programs are shown in Appendix A.

In his Turing Award lecture, Backus[Backus78] formalized the functional language called Functional Programming, or FP. It consists of :

1. A set of objects : O,
2. A set of primitive functions : F,
3. A set of functional forms : FF,
4. A set of definitions : D,
5. The application operation, :.

Objects in FP can be either atoms or lists. An atom is a finite string of digits or characters. A list is a non-empty sequence of objects. The atoms T and F are used to denote "true" and "false". A null list denoted as "()" is considered an atom. The character ? is called "bottom" and it is used to denote an error.

All FP primitive functions map objects into objects. They are formally characterized using a modification of the McCarthy's conditional expressions [McCa65]:

$$p_1 \rightarrow e_1; \ \ldots \ ; p_n \rightarrow e_n ; e_{n+1}, \qquad (1)$$

Expression $e_1$ is returned if predicate $p_1$ is true, $e_n$ if $p_n$ is true and $e_{n+1}$ if none of the predicates are satisfied.

The application operator : denotes that a function is applied to an object. For example, applying a user defined function f to an object X is expressed as f : X . As

a result, the input object X is transformed into a new object Y, i.e. $f : X \rightarrow Y$ . The following are definitions of some of the FP primitive functions and functional forms.

## FP PRIMITIVE FUNCTIONS

**Select Primitive Functions,** for a nonzero integer n:

**n : x** ≡

$x = (x_1, ..., x_k)$ and $0 < n \leq k \rightarrow x_n$;

$x = (x_1, ..., x_k)$ and $-k < n \leq 0 \rightarrow x_{k+n+1}$ ; ?

**Pick : (n,x)** ≡

$x = (x_1, ..., x_k)$ and $0 < n \leq k \rightarrow x_n$;

$x = (x_1, ..., x_k)$ and $-k < n \leq 0 \rightarrow x_{k+n+1}$ ; ?

**Last : x** ≡

$x = () \rightarrow ()$;

$x = (x_1, x_2, ..., x_k)$and $k \geq 1 \rightarrow x_k$ ; ?

**First : x** ≡

$x = () \rightarrow ()$;

$x = (x_1, x_2, ..., x_k)$and $k \geq 1 \rightarrow x_1$ ; ?

**Tail : x** ≡

$x = (x_1) \rightarrow ()$;

$x = (x_1, x_2, ..., x_k)$and $k \geq 2 \rightarrow (x_2, ..., x_k)$ ; ?

**Distribute From Left and Right**

**DistL : x** ≡

$x = (y, ()) \rightarrow ()$;

$x = (y, (x_1, x_2, ..., x_k)) \rightarrow ((y, x_1), ..., (y, x_k))$ ; ?

**DistR : x ≡**

$x = ((), y) \rightarrow ()$;

$x = ((x_1, x_2, ..., x_k), y) \rightarrow ((x_1, y), ..., (x_k, y))$ ; ?

**Append Left and Right**

**ApndL : x ≡**

$x = (y, ()) \rightarrow (y)$;

$x = (y, (x_1, x_2, ..., x_k)) \rightarrow (y, x_1, ..., x_k)$ ; ?

**ApndR : x ≡**

$x = ((), y) \rightarrow (y)$;

$x = ((x_1, x_2, ..., x_k), y) \rightarrow (x_1, ..., x_k, y)$ ; ?

**Transpose**

**Trans : x ≡**

$x = ((), ..., ()) \rightarrow ()$;

$x = ((x_1, x_2, ..., x_k)) \rightarrow (y_1, ..., y_m)$ ; ?

where $x_1 = (x_{i1}, ..., x_{im})$ and $y_j = (x_{j1}, ..., x_{kj})$,

$1 \leq i \leq k$ , $1 \leq j \leq m$

**Reverse : x ≡**

$x = () \rightarrow ()$;

$x = (x_1, x_2, ..., x_k) \rightarrow (x_k, ..., x_1)$ ; ?

**Concat : x ≡**

$x = ((x_{11}, ..., x_{1k}), ..., (x_{m1}, ..., x_{\bar{z}})) \rightarrow (x_{11}, ..., x_{1k,...,}x_{m1}, ..., x_{\bar{z}})$ ; ?

**Iota : x ≡**

$x = 0 \rightarrow ()$;

$x \in N^+ \rightarrow (1, 2, ..., x)$ ; ?

**Predicate Test Functions**

**Atom : x ≡**

$x \in Atoms \rightarrow T$ ;

$x \neq ? \rightarrow F$ ; ?

**Equal : x ≡**

$x = (y, z)$ and $y = z \rightarrow$ T;

$x = (y, z)$ and $y \neq z \rightarrow$ F; ?

**GT : x ≡**

$x = (y, z)$ and $y > z \rightarrow$ T;

$x = (y, z)$ and $y \leq z \rightarrow$ F; ?

## FP FUNCTIONAL FORMS

Functional forms are expressions that combine functions or objects into new functions. For example:

**Composition**

**( f@g ) : x ≡ f : ( g : x )**

**Construction**

$\{ f_1, \ldots, f_n \} : x \equiv ( f_1 : x , \ldots , f_n : x )$

**Condition**

$( p \rightarrow f; g ) : x \equiv$

$(p:x) = T \rightarrow f : x ;$

$(p:x) = F \rightarrow g : x ; ?$

**Apply To All**

**AP f : x** $\equiv$

$x = ( x_1, \ldots , x_n ) \rightarrow (f : x_1, \ldots , f : x_n )$

$x = () \rightarrow () ; ?$

**Insert**

**IN f : x** $\equiv$

$x = ( x_1 ) \rightarrow x_1;$

$x = (x_1, \ldots, x_n)$ and $n \geq 2 \rightarrow f : (x_1,$ IN f : $(x_2, \ldots , x_n)) ; ?$

In FP, the set of primitive functions and functional forms is used to define new functions. For example, one can define a function SIN to perform the sin of a sum of numbers x and y, that is, $\sin(x+y)$ as SIN = sin @ + . The symbol @ is the composition functional form with sin and + being its functional arguments. The composition establishes the sequence of evaluation of its arguments. Applying the SIN function to a list of two atoms x and y, SIN : (x,y) will first apply the + primitive function to the list object, and then the sin function.

An FP program is an expression written in the form of a string of functions and functional forms. There is a single input object and a single output object. The execution of each function contributes to the reduction of the program string. The program terminates when the string is completely reduced, and the result is represented by the final object. In appendix A, four FP programs, later used as benchmarks, are shown.

# 3. A Uniprocessor Implementation Of FP

A uniprocessor implementation of FP is described by first specifying the FP model of computation and then the different models of implementation. A compiler for FP is discussed and various implementation constraints are considered.

## 3.1. A Uniprocessor Model of Computation

In a Von Neumann model of computation, the state of computation is specified by an instruction pointer ( giving the address of the current instruction in the executed program ) and by a pointer to a vector of values representing the portion of memory used by the program to store local and global variables ( the program environment ). That is, a state can be described with two pointers, an instruction pointer and an environment pointer. The next instruction is obtained either by incrementing the instruction pointer or by loading the next value.

In the FP model of computation considered here, all primitive functions are treated as single non-interruptible "instructions" executed by the processor. An FP program, which is a single expression of primitive functions and functional forms, can therefore be seen as a sequence of instructions executed on the FP machine. The sequence of execution is determined by the functional forms. An example of how the condition and the compose functional forms sequence the execution of functions is shown further in this chapter.

The state of computation S of the FP model described here, is also specified with two pointers. One pointer is the FP instruction pointer, that is, the function pointer f , and the other is the object pointer o. Therefore, the state S can be described as: S = ( f,o ). The function pointer f points to the current primitive function F being executed, and the object pointer o points to the object O that the function F is applied to.

The computation of each primitive function maps a current state of the FP machine $S_1$ into a new state $S_2$. That is, we can write,

$$\text{FP function computation : } (f_1, o_1) \text{ ---> } (f_2, o_2) \qquad (2)$$

where $f_1$ points to the function $F_1$ being computed, $o_1$ the current object $O_1$ in memory, $f_2$ is a pointer to the next function $F_2$ to be computed and $o_2$ points to the object $O_2$ obtained by applying function $F_1$ to object $O_1$. The value of the next function pointer $f_2$ is obtained either by incrementing the current function pointer $f_1$ or by loading the next value. Recursive funcition calls are also handled and this is dicussed later in this chapter.

After computing function $F_1$, object $O_1$ no longer exists, and the new object $O_2$ becomes the current object in memory. This means that at each point during the execution of the FP program, there is only one object in memory and one function ( primitive or functional form ) being applied to it.

The computation of an FP program can be specified as a sequence of pairs of pointers ( f,o ), where $f_1$ points to the first function in the FP program, and $o_1$ points to the input object.

$$\text{FP program:} (f_1, o_1) \text{-->} (f_2, o_2) \text{--> ...--> } (f_n, o_n) \qquad (3)$$

The last function $F_n$ that is applied to object $O_n$ will produce object $O_{n+1}$ representing the final result.

## 3.2. A Uniprocessor Implementation Model

A model of implementation of FP on a uniprocessor is shown in Figure 1. An FP program is compiled into a string of function calls that are executed by the FP machine. How this is done is discussed later in this chapter.

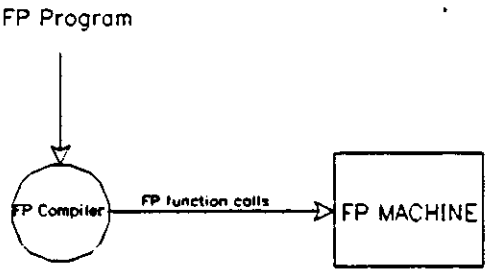


**Figure 1: A Uniprocessor Model Of Implementation**

In order to execute FP, whether on a standard off-the-shelf processor or on a special-purpose processor, a data structure must be chosen to represent FP objects in memory. Once this is done, each primitive function and functional form is implemented using the host instruction set. The data structure chosen and the implementation of the FP primitive functions and functional forms are described in detail in chapter 4.

### 3.2.1. Object Oriented Implementation Model

One of the main attributes of functional programming is the lack of variables. By always applying one function to one object, and by confining the domain of each function to the object it is applied to, functional programming is immune to possible side effects. Therefore, the execution of each function cannot affect the execution of other functions except through its output object. The result of each function is dependent only on its input object. For this reason, functional languages are also referred to as object-oriented languages.

At the execution level, strictly following an object oriented implementation, the input object would not be affected by the creation of the new output object. That is, a function applied to the input object would create a new object using the input object and not destroying it. An example of the sequence of functions FIRST @ SEL2 applied to the input list object ( 1 ( 2 3 ) ) is shown in Figure 2. The list object is represented in an abstract fashion since a more detailed analysis of the data structures used will be given later in this report.

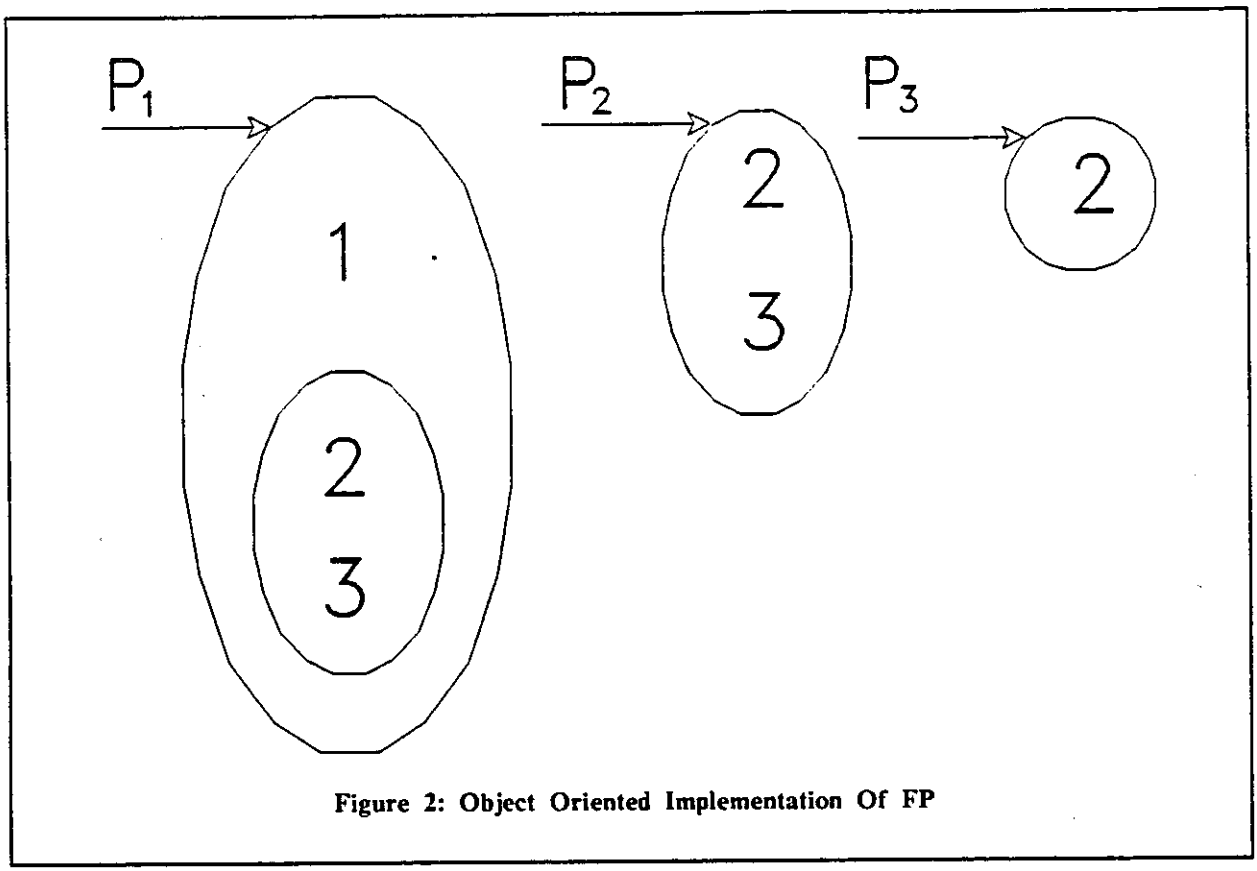


**Figure 2: Object Oriented Implementation Of FP**

Pointer $P_1$ is the pointer to the input object ( 1 ( 2 3 ) ). After applying function SEL2 the new pointer $P_2$ points to the list ( 2 3 ) and after selecting the FIRST element, the final object pointer $P_3$ points to atom 2.

One can note that for most of the functions used, the described implementation would lead to redundant copying of parts of the input object in order to create a new object. In the simple example shown in Figure 2 both function applications resulted in copying parts of the input object. The amount of redundant copying is especially high for those functions that just rearrange the elements of an input object. In these cases the redundancy would be equivalent to copying the input object in order to form the output object.

Therefore, an implementation like this would cause a significant overhead that would not be part of the computation of the FP program. In order to avoid the redundant copying of portions of the input object, an alternative model of execution is described.

### 3.2.2. Pointer Oriented Implementation Model

Instead of literally creating a new object every time a function is executed, one can pass to the function a pointer to the current object. The function will perform the operation on the object pointed to by the pointer it had received as a parameter. This approach would be identical as if we were to pass the input object as a "VAR parameter", ( passing parameters by reference ), to the calling function.

By executing several functions, one after another, the input object goes through a series of transformations. Each function will reuse its input object as much as possible in order to form the output object. An example of this approach is shown in Figure 3 using the same sequence of functions as in Figure 2 .

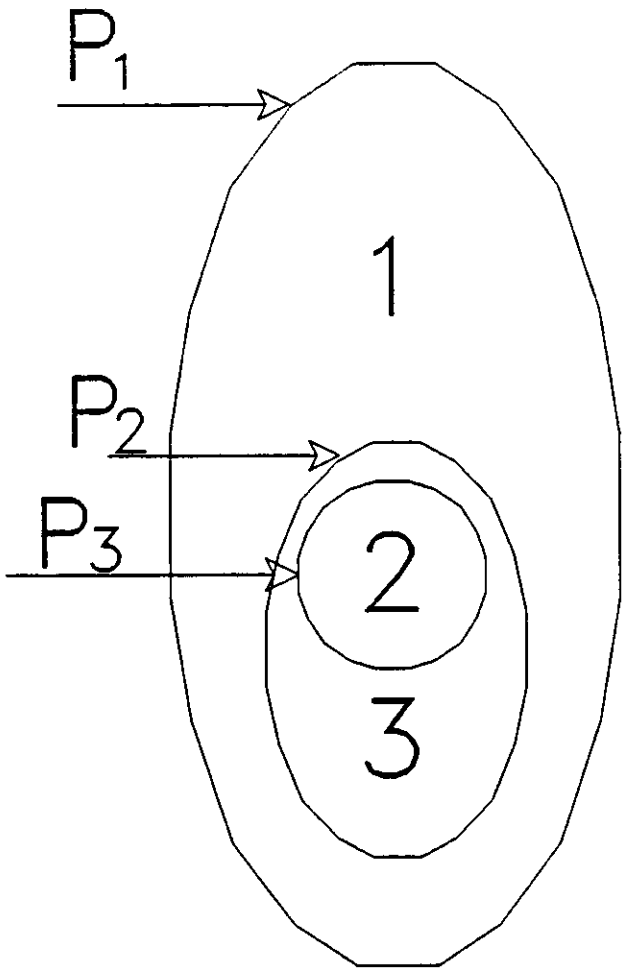


**Figure 3: Pointer Oriented Implementation Of FP**

One can note that the pointers $P_2$ and $P_3$ point to different portions of the original input object and that there has been no redundant copying. Portions of the old object that are not reused in the new object are discarded as garbage, ( and will be collected later in this report ). Because this approach reuses the input object as much as possible, it is more "economical", as far as memory resources and performance are concerned, than the object oriented implementation.

It should be clarified here, that even though the objects are passed by reference, this is done only at the implementation level, and there is still no possibility of side

effects. At the level of the language, functions are still applied to only one object at a time, and the domain of each function is only the object it is applied to. Therefore, the only thing that is done here is to avoid the unnecessary copying of objects that would account for a significant implementation overhead. The implementation model that is used from here on is of the pointer approach.

It is interesting to note that even though the object oriented approach performs so much unnecessary copying, it has one nice feature, that is, it leaves behind a "trail" of old objects. If one was to store the pointers to these objects in a "cache", one could reuse some of the old objects without having to compute them. Also, by storing these pointers one could implement a "program roll back" facility that could be a powerful debugging tool.

## 3.3. Interpretation Vs Compilation

An FP program is represented as a string of calls to primitive FP functions, functional forms or user defined functions. The primitive functions are directly executable by the FP machine, whereas the functional forms and the user defined functions result in further function calls.

In most implementations of FP on a uniprocessor architecture, FP is interpreted. In the Lahti intepreter [Lahti80], the FP program is represented in memory as a string of characters. The interpreter parses the string from left to right and calls the functions as they are encountered. This is an example of a string reduction approach to an FP implementation. The benefits of having FP intepreted are in the interactive environment which is more amenable to debugging. The loss is in the speed of execution.

As opposed to an interpreted string-reduction approach, FP could be compiled ( assembled ), and stored in memory as a sequence of function calls. This would include calls to functional forms which would control the sequence of execution. For example, the compose functional form "@" of functions f and g defined as:

$$f \ @ \ g : X \ = f : ( g : X ) \qquad (4)$$

can be implemented as a function that has as parameters functions f and g, and the pointer "object-pointer" to the current object in memory. That is,

```
function Compose( f, g, object_pointer);
begin
  object_pointer := g(object_pointer);
  object_pointer := f(object_pointer);
  return( object_pointer );
end;
```

The compose functional form will first apply function g to the object pointed to by the object__pointer returning a pointer to the new object. This pointer is then the new object-pointer, and it is passed to function f. Therefore, the composition of the two functions f and g will return a pointer to the object derived after applying functions g and f to the input object. One must note that the same "object__pointer" is used in each function application. Each function returns a pointer to the new object which is then regarded as the new "object__pointer". In order to cause no confusion in the specification of FP functions, we can implicitly define that each function is always applied to the object pointed to by the object-pointer, unless otherwise specified. We can also always presume that each function returns a pointer to the new object created, and not specify it . In this case the function Compose is simply written as:

```
function Compose( f, g );
begin
  g;
  f;
end;
```

Similarly, the condition functional form of functions f, g and h is applied to the input object pointed to by the object__pointer. Function f is first applied to the input object and IF this results in a true boolean value THEN function g is applied to the original object ELSE h is. This means that a copy of the input object has to be made before the first function is applied to it ( otherwise it is lost ). To do this, a "copy__object" function is implemented ( this is described in detail in chapter 5 ). It takes an object pointer, copies the object it points to and returns a pointer to the new object. That is,

```
function Condition( f, g, h );
x: local_var_in_register;
begin
  x = copy_object;
  if f(x) then
     g;
  else
     h;
end;
```

After implementing the functional forms, an FP program is assembled into a sequence of function calls to primitive functions, user defined functions or functional forms. In the following example the InnerProduct program that performs the inner product of two vectors is written first in FP and then in the pseudo Pascal notation:

```
FP:
       InnerProduct = AP + @ IN * @ DISTL;

Pseudo Pascal
       Program InnerProduct;
       begin
          compose(AP +, compose( IN *, DISTL );
       end;

AssembledCalls
     label1: DISTL;
     label2: IN * ;
     label3: AP + ;
```

Even though the above example depicts a simple FP program being assembled into a series of function calls, it offers an idea of how a compiler might resolve some of the calls made within the functional forms ( in this case the compose functional form ). What we have really done here is simply "decompose" the FP program into a sequence of calls. Since each function call has a label, recursive calls may be handled as well as calls to user defined functions. In order to support recursive calls, a control stack is assumed in memory. A compiler for FP has been implemented and is described in [Feller81] and [ShihLi84].

## 3.4. Uniprocessor Implementation Constraints

Even though claims are made that functional languages increase the programmer's productivity, compactness of coding, programming verification and debugging, at the implementation level, there are still problems that have earned functional languages a reputation of running slowly on uniprocessors. Some of the issues discussed here are:

1. Inefficiency due to object replication.

2. Inefficiency due to redundant computations.

3. Most of the implementations have been interpretive.

4. Use of inappropriate data structures

5. Problems in Memory Management

The issues of interpretation versus compilation have already been discussed and the constraints encountered in memory management will be discussed in detail in chapter 5.

### 3.4.1. Replication Of Objects

FP offers constructs that manipulate list structures in parallel, allowing for concurrent computations in a multiprocessor environment. Some of these functions like the Construct and Condition functional form or the Distribute primitive function, require replication of the input object. In the case of the Construct functional form the object is replicated as many times as there are functions in the construct whereas in the case of the condition, a single replication is required. The Distribute left ( right ) primitive will replicate the first ( second ) element of the input list as many times as there are elements in the second ( first ) list, ( see chapter 2 for definitions of these functions ) .

Besides the replication of objects that stems from the definition of some of the FP functions, there is the replication due to the nature of functional languages. That is, since there are no variables in FP, one can not reference different parts of an input object and use them later in the computation. Rather, parts of the object that will be needed later in the program have to be carried through the computation as

part of the current object in memory. In the following example, we apply a construct of the FIRST and LAST functions to an n element object X.

```
+ @   FIRST,LAST  : (xn , . . . , x1) = X
                  -> + : ( FIRST : X , LAST : X )
                  -> + : (xn , x1)
      An Example of Object Replication
```

From this simple example we can notice two things. First the object X was replicated twice, and second, in each case the object X was transformed into a new object so that the original object X is no longer available to future computations.

Let us now look at a string of functions applied to the result obtained after applying the construct.

$$n@m@k@...@h@g@\{FIRST, LAST\}:X \qquad (5)$$

If at any point in the computation we want to reference any portion of the object X we have to modify the program. For example, in order for function m to be applied to its input object with the "FIRST of X" appended to the left, the following changes would have to be made:

$$n@m@\{FIRST, k@...@h@g\{FIRST, LAST\}\}:X \qquad (6)$$

Since the result of the first application of the function FIRST to the object X could not have been referenced later in the program, a construct functional form had to be introduced, which inevitably leads to further object replication.

A further analysis of object replication and its implication on the implementation of FP is beyond the scope of this report. It is just recognized here that object replication does account for a significant overhead in the implementation of FP on a uniprocessor.

### 3.4.2. Redundant Function Application

Besides the fact that object replication seems to be inherent to the functional programming style, another undesired effect is the often redundant computation that

is performed. In the previous example, we saw the redundant computation of FIRST : X. In the following example,

$$\{1,\text{tail} \circ 1, \text{last} \circ 1\} : X \tag{7}$$

we can see that, regardless of the fact that object X is replicated 3 times, the actual operations that will be performed are :

$$1{:}X \tag{8}$$

$$\textit{tail}\circ 1{:}X \tag{9}$$

$$\textit{last}\circ 1{:}X \tag{10}$$

In all three cases the first operation is the same, and it will be computed 3 times, even though it would be useful to save the result for later function applications. Such examples are quite common in FP programs and are particularly obvious in the case of recursive algorithms.

One of the main advantages of functional programming languages is the property that a series of functions applied to an object may be simply replaced by the resulting object. This means that regardless of where or when a certain function is called, given the same input, it will always produce the same output object. This property called **referential transparency** ( see introduction ), would allow us to replace any occurrence of an FP function applied to the same input object, with the resulting output object.

It is an open question how to use this property in a uniprocessor implementation. A possibility is to precompute the number of times a certain function is applied to the same object, replicate the output object and use it when the same combination of function and object is encountered during program execution. One would have to be able to label and differentiate objects in memory at compile time. This, though, would be difficult to do without omitting instances where the same objects would be labeled differently, ( for example in recursive algorithms ).

### 3.4.3. The Use Of Inappropriate Data Structures.

It was mentioned earlier that an important step in the implementation of FP ( whether in a uniprocessor or multiprocessor environment ), is to chose an appropriate

data structure to represent objects in memory. Several aspects must be taken into consideration. First, most FP primitive functions are standard list manipulating operations. Data structures that are most suitable for each operation are well known and documented [Knuth73]. Nevertheless, choosing a data structure to represent FP objects should be made to reflect the nature of the language ( consisting of primitive functions and functional forms ), and the programmer's use of the language. One should also consider implementation constraints and overheads.

It should be noted here that choosing an appropriate data structure is machine dependent, that is, it depends on the hardware support available. This is further discussed in chapter 4.

# 4. The FP Data Structure

Implementing FP on a processor consists of coding the FP primitive functions and functional forms using the processor's instruction set. In order to do so, a data structure must be chosen to represent FP objects in memory. FP objects are basically list structures whose elements are either lists or atoms, ( see chapter 2 ). Once a data structure has been adopted each FP function is implemented to perform its logical function according to the FP definitions ( see chapter 2 ).

FP primitive functions are in fact list manipulating functions and choosing an optimal data structure for each of the functions is a well known and documented issue [Knuth73]. For example, a linked list or a pointer data structure is more suitable for any function that performs insertion or deletion of elements of a list whereas an array type of data structure is more suitable for selecting elements of a list. It is our concern to chose a data structure that will reflect not only the nature of the FP functions but also the way in which the user uses the FP functions. Two different data structures are considered here, a Pointer and a Sequential data structure.

## 4.1. A Pointer Data structure

There are two basic building blocks used in the pointer representation, an Atom block and a List block. Both blocks have a tag field used to distinguish the blocks and to store any information needed for memory management. The size of the tag is not important at the moment, and depends on the amount of information necessary to keep. The list block has two pointer fields while the atom block consists of a pointer field and a value field. The two pointer fields in the list block point to the First element in the list and to the lists' Next element. In Figure 4 we show both building blocks and a representation of a list object.

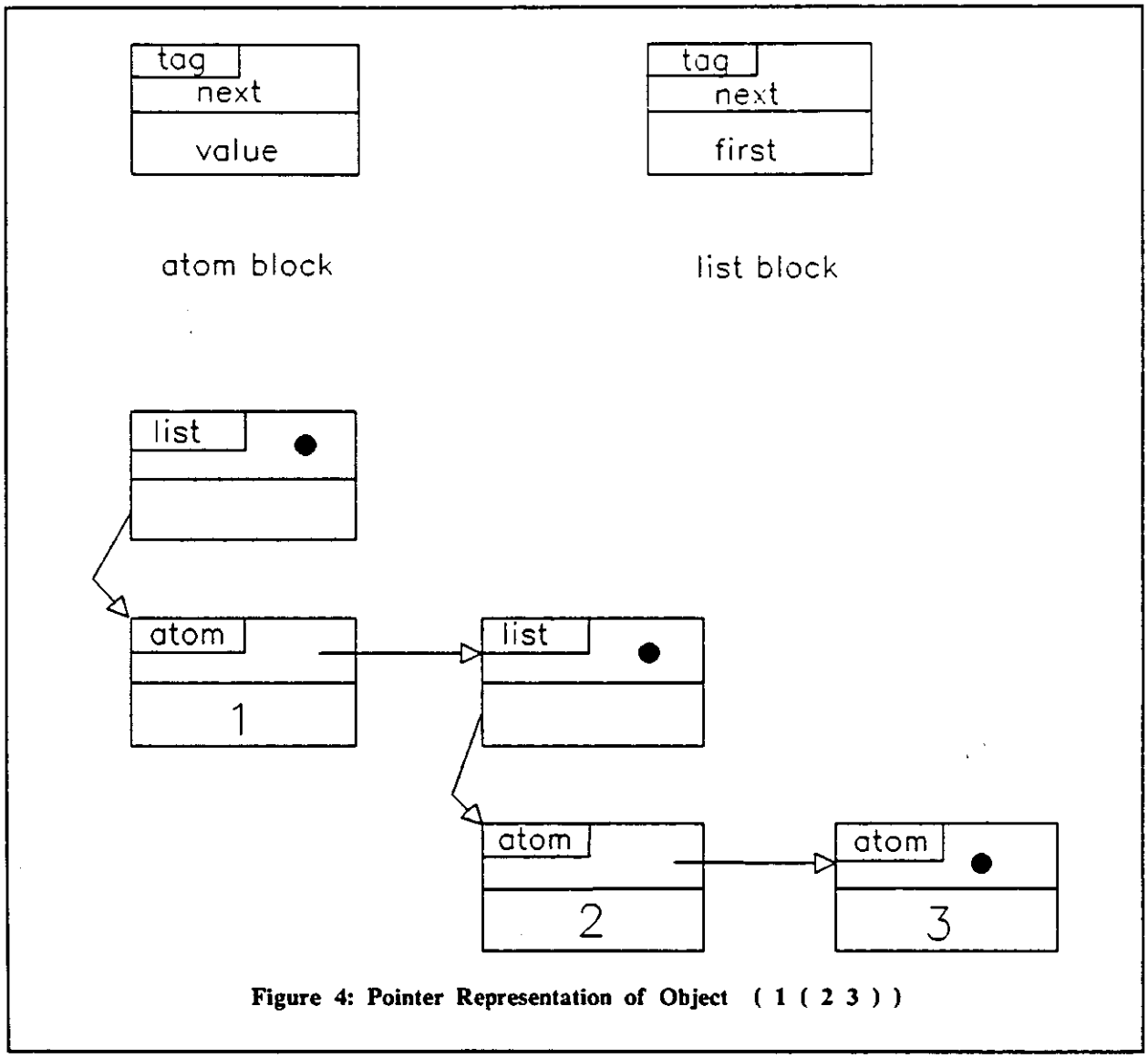


**Figure 4: Pointer Representation of Object ( 1 ( 2 3 ) )**

## 4.2. The Sequential Data Structure

In the Sequential data representation the blocks used are similar to the ones used in the Pointer representation. The ordering of the elements of a list is implicitly defined here to be sequential. In this case, the pointer fields of the List block point to the First and Last element of a list and since the location of the Next element is implicitly defined, the Atom block contains only a Value field. Both blocks are shown

in Figure 5 together with the representation of a list structure.

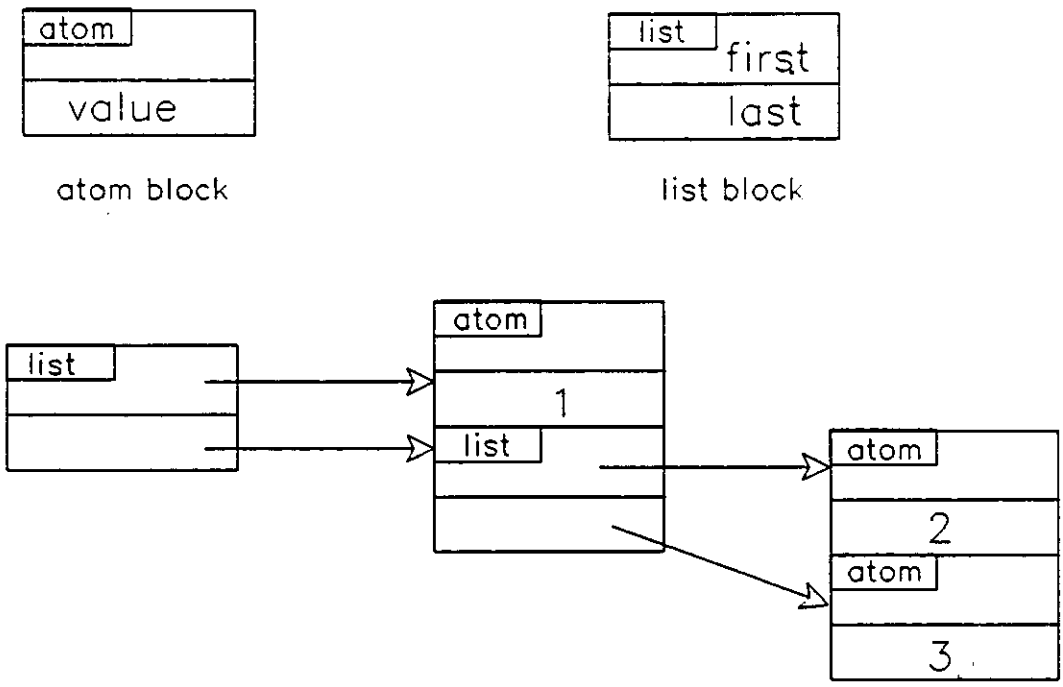


**Figure 5: Sequential Representation of Object ( 1 ( 2 3 ) )**

In the actual implementation, a design choice was made to have both blocks of equal size. This means that instead of the atom block having only one value field, it is represented as having two fields, one of which is being used. Such a choice was made because the main advantage of the sequential data structure lies in the ability to calculate the location of an element in a list knowing only the location of the first element. From Figure 5 one can note that if the size of the list header differed from the atom block, one could not exactly calculate the location of an element in a list.

## 4.3. Pointer Vs Sequential Data Structure

In order to determine which data structure is more favorable for the implementation of FP we need to know more about the use of primitive FP functions from the programmer's point of view. In tables 1a, 1b, 1c and 1d we show the dynamic frequencies with which the primitive functions occurred in the four benchmarks given in Appendix A. The results from the Matrix Multiplication benchmarks are given as a function of the size of the matrix n. The results from the Insertion Into An Ordered List benchmark depend on the length of the list L but are shown here as a percentage of the overall functions executed. Because the results from the Quicksort and the Sieve benchmarks depend on the actual data, we present an average over a variety of data sets.

| *Functions* | *MM(n) Benchmark* |
|---|---|
| *Plus* | $n^2(n-1)$ |
| *Times* | $n^3$ |
| *Sel1,2,3* | 2 |
| *DISTr,l* | n+1 |

| *Functions* | *IN(L) Benchmark* |
|---|---|
| *GT* | 9% |
| *Tail* | < 9% |
| *Sel1,2,3* | 66% - 73% |
| *Rev* | < 9% |
| *APr,APl* | < 9% |
| *MOD* | 6% |
| *CONstant* | < 3% |

Tables 1a,1b: Dynamic Frequencies of Primitive FP Functions
For the MM and IN(L) Benchmark

| *Functions* | *Sieve(10) Benchmark* |
|---|---|
| *Null* | 7% |
| *Tail* | 7% |
| *Sel1,2,3* | 55% |
| *EQ,NOT* | 12% |
| *Rev* | 1% |
| *APr,APl* | 5% |
| *CONstant* | 7% |
| *MOD* | 6% |

Table 1c: Dynamic Frequencies of Primitive FP Functions
For the Sieve(10) Benchmark

| *Functions* | *Quick(12) Benchmark* |
|---|---|
| *Null* | 6% |
| *Tail* | 4% |
| *Sel1,2,3* | 54% |
| *GT* | 3% |
| *DISTr,l* | 2% |
| *APr,APl* | 3% |
| *CONstant* | 8% |
| *LN,CONC* | 4% |
| *ID* | 8% |
| *Div,Split* | 8% |

Table 1d: Dynamic Frequencies of Primitive FP Functions
For the Quick(12) Benchmark

One can note the high usage of the Select functions in these FP programs. This is due to the fact that FP does not have variables and is thus unable to reference parts of an object that have been used earlier in the program. The Select functions are used to select portions of an object, especially if they have to be saved for future use in the program. One can also note that all of the select functions selected one of the first three elements of a list.

Nevertheless, the four benchmarks observed do not offer a complete insight into the nature of FP so one should not draw conclusions without making a more thorough analysis. One can particularly notice that the benchmarks do not represent well

enough the variety of primitive functions in FP and that many of the functions are represented only in one benchmark. A complete analysis should include more benchmarks and a greater variety.

One can though, observe what is intuitive, and that is that the most frequent functions are the Select primitives. Based only on this observed property, one could conclude that a sequential representation could be better since the cost of selection would be constant as opposed to linear in the case of the pointer representation. Nevertheless, in cases where the selection constant is small, the advantage of the sequential representation over the pointer would not be significant at all. Therefore, this should not be a deciding factor in determining which data structure to chose.

Let us look at how another primitive function could be implemented, for example the Concatenate function. In the sequential representation, one would have to create a new list with all the elements of the list in consecutive memory locations. Ignoring ( for the moment ) any possible memory conflicts, one would have to move the second list so that its first element is directly beneath the last element of the first list. If the second list is large, this could be an expensive operation. In the pointer representation, concatenating two lists does not require any moving of blocks at all. One has to just change the value of the next pointer in the last element of the first list to point to the first element of the second list. Nevertheless, to get to the last element of the first list, the list has to be traversed. It so happens that traversing a list requires going into memory and following a pointer value, which is as costly as moving a stored value from one memory location to another.

In effect, if the length of the first list is greater than the second, the sequential representation would be better than the pointer representation, especially since many processors today support fast moving of blocks of memory from one location to another. Again, based on this example, one cannot claim a strong preference for one of the proposed data structures because it is data dependent.

Since following a pointer through memory is similar in cost to moving a stored value from one memory location to another, other primitive functions like Reverse, Transpose and Distr and Distl do not offer a clear direction as to which data structure one should implement.

In the case of the Length and the Appendr functions, it is true that the sequential representation is better, since it does not require traversing the list. On the other hand Tail, Appendl, First, and all the binary operations perform similarly in both cases.

Therefore, based only on the cost of implementation and given a vague idea of the frequency of use of each primitive function, one cannot claim a distinct advantage of one representation over another. Another factor has to determine the design choice. This factor is the global memory management scheme that would support each data structure and it includes allocation of memory cells and the collection of cells discarded after each function application. This issue is discussed further in the following section.

## 4.4. Memory Management Issues

Under memory management we are primarily concerned with two major issues. First, allocating free cells to the FP primitive functions and functional forms, and second, reclaiming unused storage cells and making them accessible to the FP program.

Representing FP objects in memory using a Pointer data structure makes no demands as to where the list and atom blocks have to be. This generality of cell location leads to a simple memory allocation algorithm. Whenever a primitive function or functional form needs to allocate more cells, they can be allocated from any place in memory without any constraint.

On the other hand, the Sequential data structure lacks the generality of the Pointer representation but enhances the performance of some primitive FP functions like Select, Last, Split and others. Nevertheless, there is a price to pay in order to maintain a data structure that has elements of a list in consecutive memory locations. For example, let us consider a simple case of the AppendR function being applied to a list structure (( 1, 2, 3 ), 4 ). In Figure 6 on page 30 we show a possible memory representation of the input object using the Sequential data structure. The result is a list of atoms representing object ( 1, 2, 3, 4 ).

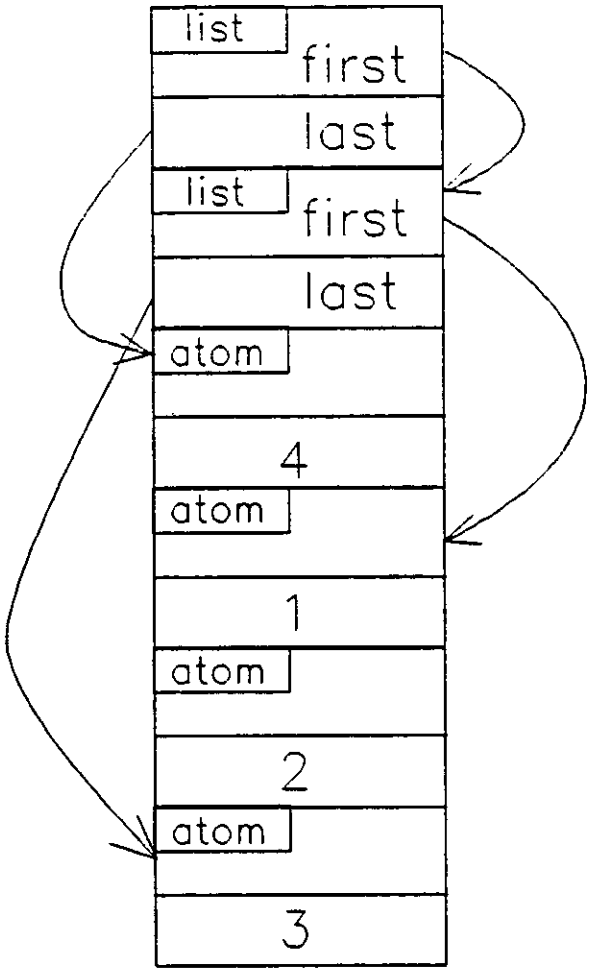


**Figure 6: Representing Object ( ( 1 2 3 ) 4 ) In Memory**

One can note in this example that the cost of implementing the AppendR function depends on whether it is possible to allocate a block for the atom '4' directly below atom '3'. A memory management scheme supporting the Sequential data structure implementation would have to be able to detect such conflict situations. Resolving such a conflict would consist of forming a new object somewhere else in memory where there is sufficient space. It is quite common for FP programs to form very large objects in memory so that any copying could lead to a significant overhead.

Another problem is that the memory management algorithm supporting a Sequential data structure would have to determine if an "expanding" object in memory is about to "overlap" an already existing portion of an object. ( for example, in case of such a conflict, concatenating two lists could lead to copying both lists into a sufficiently large space of memory ). This means that the memory management algorithm would have to maintain a table or a map of the memory usage.

Performing Garbage Collection for a Sequential data structure implementation would require a header for each block of cells in memory. The header would contain the size of the block and a pointer to the next available space in memory. One could get into a situation where memory gets fragmented so that no contiguous area can be found to accommodate a certain large object, even if there is memory available. This would then lead to more frequent garbage collection and greater implementation overhead.

Using the Pointer data representation new blocks may be allocated until there are no more blocks left, so that "premature" garbage collection due to memory fragmentation can not occur. In addition, many different algorithms exist for performing garbage collection of cells that were part of list data structures that use pointers, exist [Cohen81]. Most of them are simple, and require an extra 2 or 3 bits to be reserved in the tag of each cell.

It is interesting to note that the sequential policy has an advantage over the pointer data structure in terms of the number of times the memory allocation routine must be invoked. In the pointer case, it is for every single cell, but in the case of the sequential data structure one could allocate only once a large enough block for all of the elements of a list to fit. That is, by subtracting the address of the first element in the list from the address of the last, one can know precisly the number of cells in the list. If this list is being copied to another location in memory, only one memory allocation call is made, requesting the correct amonunt of memory.

Implementing a memory allocation and garbage collection algorithm for the Pointer data structure is simpler than for the Sequential data structure. It avoids premature garbage collection due to memory fragmentation and possible complications due to an object "expanding" in memory.

One could use a separate Memory Management Coprocessor to support the choice of a Sequential data structure. However, in the case considered here, that is, a uniprocessor implementation, the Pointer data structure is a more suitable choice.

Different memory allocation and garbage collection policies to support the Pointer data structure are considered in the following chapter.

# 5. FP Memory Management

In order to manage the pointer data structures used in the implementation of FP, three different memory management approaches are considered:

1. Sequential

2. Linked List

3. Stack

In the first two approaches the garbage collections are implementations of the static mark and sweep algorithm [Knuth73], [Cohen81]. This means that the FP execution has to be halted whenever end of memory is reached and the unused cells have to be collected. The third method uses a stack to store pointers to garbage data structures left behind after a function is applied to an object. New cells may be allocated either from this stack or from the available memory that has not yet been allocated. To do this, the primitive FP functions have been modified so that the garbage pointers are pushed onto a stack during their execution. Therefore this method is the only one considered here that is truly dynamic. All three algorithms have been implemented (simulated), and overheads associated with each implementation are discussed. Matrix Multiplication and Quicksort are benchmarks used to compare the performances. We are particularly interested in using programs that require several times the available size of memory, so that garbage collection has to be performed.

## 5.1. FP Memory Management Simulator

The FP memory management simulator is implemented using the Lahti FP interpreter [Lahti80]. The interpreter represents the FP program and the input object as strings of characters. It then performs string reduction of the FP program, calling primitive functions and functional forms. The execution of these functions consists

of manipulating the string of characters that represent the input object, thus forming a new object.

The FP memory management simulator is built into the FP interpreter. Memory is represented as an array of cells consisting of two pointer fields and a tag field. The input string, entered by the user, is parsed and "created" in memory using a Create_Object routine ( see Appendix B ). The input is represented in memory using the pointer data structure. In Figure 7 an input object is created in memory using the Create_Object routine.

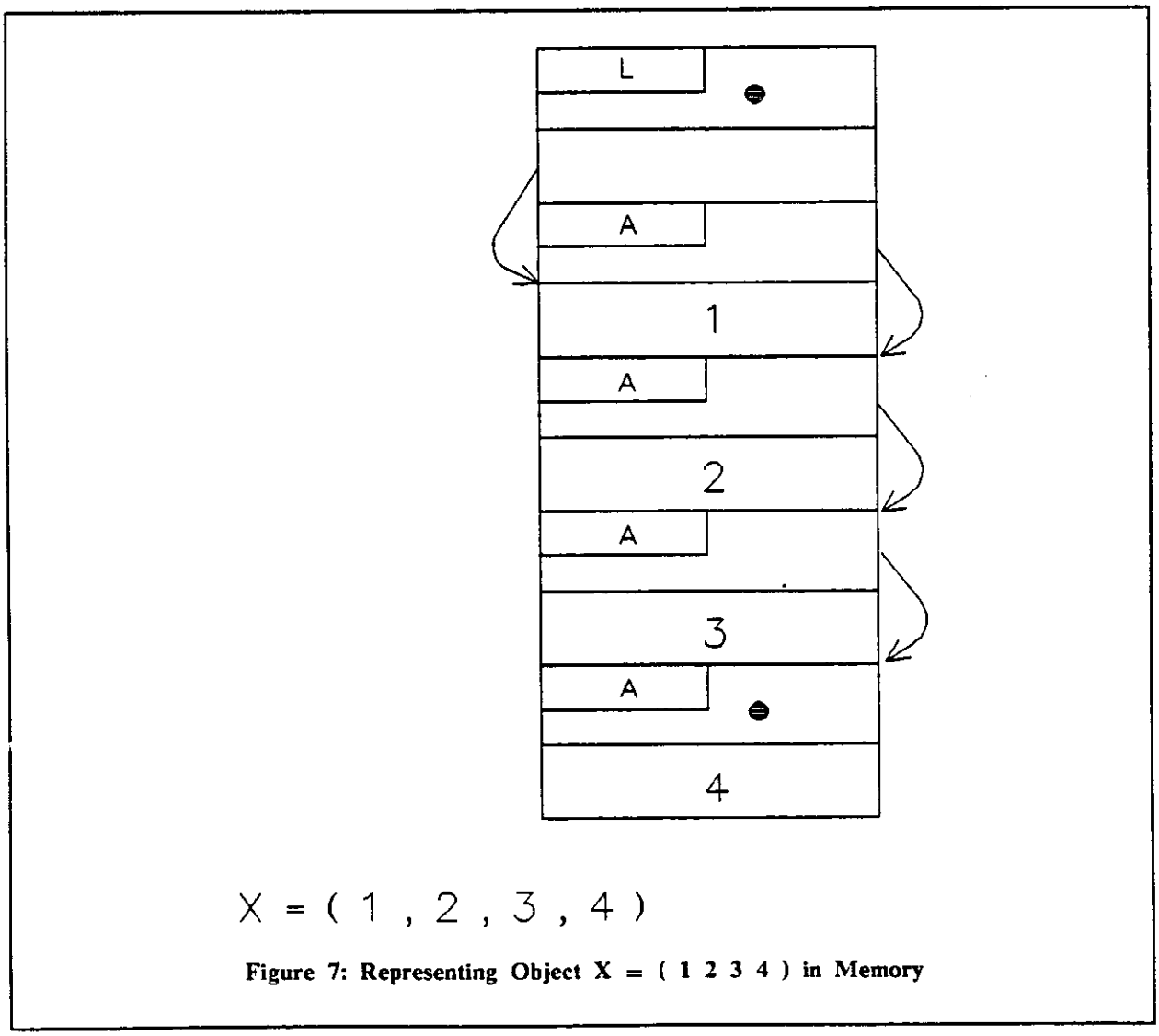


**Figure 7: Representing Object X = ( 1 2 3 4 ) in Memory**

As the interpreter calls each primitive function or functional form, it first executes its own code, and then a set of instructions that manipulate the pointer data structure, performing the same function. In figure 7.1, an example of how the LAST primitive function is first executed in the string interpreter, and then in the simulated memory array structure is shown.

```
CASE LAST:

NumOfObj = Getobjects(r);          ;r = num of objects
CopyStr(r,objptr<NumOfObj - 1>); ;Copy last element

                                   ;P is pointer to object
P = P@.value;                      ;P points to first elem.
WHILE ( P NOT = NULL )
   P = P@.next;                    ;P follows the Nextp
                                   ;P points to last elem.
BREAK;
```

Figure 7.1 LAST primitive function

Each primitive function or functional form in the simulator requires only a pointer to the current object. After performing the necessary operation, a pointer to the new object is returned. At any point in the simulator, the current object can be displayed using the Display_Object routine ( see Appendix B ). This facility played a major role in the writing and debugging of the memory management simulator. At each point of the interpretation, the interpreter's string representation of the current object in memory was displayed and compared with the data structure in simulated memory.

Regardless of the memory management policy simulated, every FP function that requires a new cell during its execution, calls an Allocate_Cell routine which returns a pointer to a free cell. Depending on which memory management algorithm is simulated, cells are allocated from different memory locations, and garbage collection is performed accordingly. In Figure 8, an example of how a cell, pointed to by cell_pointer, allocates a new cell pointed to by its 'next' field, is shown.

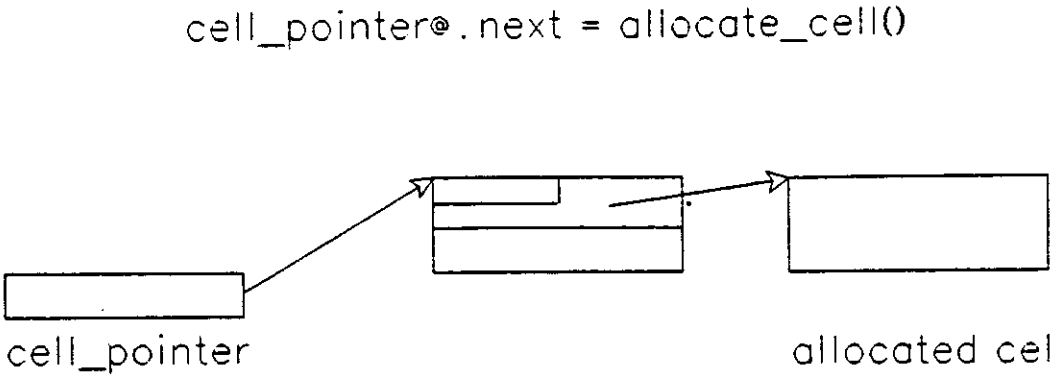


**Figure 8: Cell Allocation**

The memory management simulator allows for many useful measurements to be made. For example, one can display and observe the usage of memory after each function application. Statistics may be gathered specifying the number of times objects are copied and the total number of cells copied. Every time garbage collection is invoked, performance measurements are recorded. Many other useful parameters relating to memory management routines are computed and displayed after program termination.

## 5.2. Sequential Memory Management

In the Sequential Memory Management policy memory is allocated from a contiguous pool of available cells. Allocating a cell requires only to increment a pointer. When the end of memory is reached, the execution of the FP program has to be interrupted in order to collect the unused cells. These cells are collected by traversing and marking the data structures currently in use and reallocating them to the beginning of memory. Memory allocation continues from the newly established contiguous pool of available cells.

## 5.2.1. Sequential Memory Allocation

The sequential memory allocation algorithm is shown in figure 8.1 . Each call to the cell allocation routine will merely increment a free cell pointer. If the end of memory is reached the Maxreached flag is set.

```
ALLOCATE_SEQUENTIALLY()
BEGIN
   free_cell_pointer = free_cell_pointer + 1;
   IF ( free_cell_pointer = maxmem )
      maxreached = 1;
END;
```

Figure 8.1 Sequential Allocation Algorithm

## 5.2.2. Garbage Collection

In the memory allocation routine, the flag Maxreached is set whenever the end of memory is reached. This flag is tested in the interpreter after the execution of each primitive function or functional form. If it is set, garbage collection will be performed in two phases:

1. Identifying the storage space that may be reclaimed.

2. Including the reclaimable space into the memory area available to the user.

The first phase of garbage collection is performed by marking all the cells in memory that belong to the input object. The unmarked cells are thus reclaimable. The recursive mark algorithm is shown in figure 8.2 . Note that the actual marking consists of setting a mark bit in the tag field of each cell. The cell tag also has a bit that is used to distinguish between atom and list cells.

```
MARK(cell_pointer)
BEGIN
   IF ( cell_pointer@  NOT = NULL ) BEGIN
      cell_pointer@.tag.mark_bit = 'Marked';
      IF ( cell_pointer@.tag.atom_bit = LIST )
         MARK( cell_pointer@.value );
      MARK( cell_pointer@.next );
   END;
END;
```

Figure 8.2 Mark Routine

The cell_pointer passed to the mark routine should point to the current object in memory. If a list cell is marked, the two list pointers must be traversed, and in case it is an atom only the next pointer is followed.

The second phase of garbage collection consists of compacting all the marked cells in one end of memory. The rest of memory is then made available to the memory allocation routine. There are various types of compaction algorithms [Cohen81]. In the algorithm described in [HarEva64] and [Cohen67], memory is scanned twice. In the first scan, two pointers are used, each starting from different ends of memory. The top pointer ( at low address ) is incremented until an unmarked cell is encountered. The bottom pointer is then decremented until it points to a marked cell. The marked cell is then moved to the new unmarked cell, and a pointer to the unmarked cell is saved in the old cell. At the same time the mark bit in the copied cell is turned off. This process is repeated until the two pointers meet. By then the marked cells would be compacted into the top of memory. It is then necessary to scan the compacted area and readjust the pointers that point outside this area. The correct pointer values were stored in the old cells that were copied in the first pass. The disadvantage of this algorithm is that its time performance is proportional to the size of memory.

In order to avoid the two passes through memory, the compaction algorithm considered here uses a copying algorithm. The Copy_Object routine shown in figure 8.3 traverses the object pointed to by the current object pointer, allocates a cell for every cell encountered, and copies the TAG, NEXT and VALUE fields. By setting the free_cell_pointer of the memory allocation routine to zero before calling the Copy_Object routine, the current object is copied to the beginning of memory. However, If while copying the object, we allocate a cell that belongs to the object being copied, the result would be incorrect.

```
COPY_OBJECT(cell_pointer)
BEGIN
  IF (cell_pointer NOT = NULL) BEGIN
     p = ALLOCATE_CELL();
     IF (cell_pointer@.tag.atom = LIST) BEGIN
       p@.tag.atom = LIST;
       p@.value = COPY_OBJECT(cell_pointer@.value);
       END
     ELSE BEGIN
       p@.tag.atom = ATOM;
       p@.value = cell_pointer@.value;
       END;
     p@.next = COPY_OBJECT(cell_pointer@.next);
     RETURN(p)
     END
  ELSE RETURN(NULL);
END;
```

Figure 8.3 COPY_OBJECT Routine

In order to detect this before the copying is performed, the mark routine has to be slightly modified. That is, while the current object is being marked, a counter is incremented with each cell encountered. If we are copying the object to the beginning of memory, the count represents the location where the traversed cell will be copied to. Therefore, for each cell, we compare the count value to its current address in memory. If, for all cells, the address is greater than the count, the object may be copied. It should be noted that the cell address and the count value must be compared for every cell in the traversed data structure. This is because any of the cells could conflict with the object being copied to the beginning of memory. The modified mark routine is shown in figure 8.4 .

```
MARK(cell_pointer)
BEGIN
   IF ( cell_pointer@  NOT = NULL ) BEGIN
      cell_count = cell_count + 1;
      IF cell_pointer > cell_count THEN copying_safe;
      cell_pointer@.tag.mark_bit = 'Marked';
         IF ( cell_pointer@.tag.atom_bit = LIST )
            MARK( cell_pointer@.value );
         MARK( cell_pointer@.next );
      END;
END;
```

Figure 8.4 Modified Mark Routine

A flag copy__safe is set if no conflicts occurred during marking. If at least one conflict exists, memory is searched for a contiguous area of free cells, greater or equal to the size of the object to copy. The size of the object is equal to the cell__count obtained during marking. Searching consists of serially scanning memory from the beginning, looking at two consecutive occurrences of marked cells separated by at least one unmarked cell. The function Contiguous__area__found is true if such an area is found. In this case, the object is first copied there, and then to the beginning of memory.

If a single contiguous area is not found, the object is copied to different contiguous areas, starting from the largest area computed within the Contiguous__area__found function. This is repeated until the object may finally be copied to the beginning of memory.

The Garbage__Collection algorithm is shown in figures 8.5 . The Contiguous__area__found function and the Reallocate__iteratively routine are shown in Appendix B.

```
GARBAGE_COLLECTION()
BEGIN
   MARK(current_object);
   IF copying_safe THEN BEGIN          ; case 1
      free_cell_pointer = beginning_of_memory;
      COPY_OBJECT(current_object);
      END
   ELSE BEGIN                          ; case 2
      IF ( CONTIGUOUS_AREA_FOUND )  BEGIN
         free_cell_pointer = contiguous_area;
         COPY_OBJECT(current_object);
         free_cell_poinetr = beginning_of_memory;
         COPY_OBJECT(current_object);
         END
      ELSE REALLOCATE_ITERATIVELY;     ; case 3
   END;
END;
```

Figure 8.5 GARBAGE_COLLECTION Routine

In Figure 9 we show the three possible cases that can occur when an object is to be reallocated to the beginning of memory.

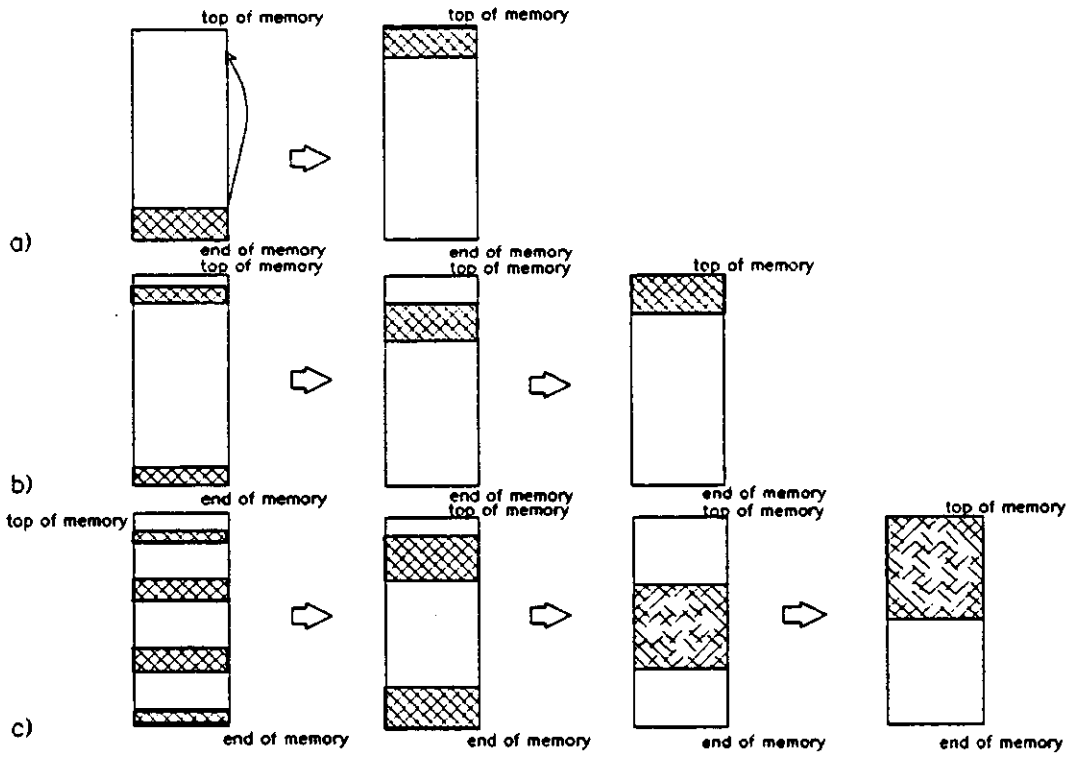


**Figure 9: Three Possible Cases of Object Reallocation**

It should be further clarified why the copying approach to compaction was considered here, as opposed to the two pointer approach. First, even though the former could lead to unsatisfactory performances if the object has to be reallocated several times to different parts of memory before it is finally copied to the beginning of memory, one could expect this case to occur less frequently as the size of memory increases. On the other hand, the performance of the two pointer algorithm is directly proportional to the size of memory.

Second, the pointer algorithm might be more appropriate for languages that create many useful objects in memory ( Lisp for example ). Copying each one of them might be unacceptable. In the implementation of FP considered here, there is always only one object in memory. Also the FP primitive functions and functional forms have been implemented with the objective to reuse as much of the input object as possible while forming the output object. Even though FP objects can grow large in

size, the actual size of the object that will be copied when garbage collection is invoked could vary significantly. For small objects copied, the overhead will be small. If the size of the copied object increases, so does the overhead. In the two pointer approach the overhead will always be significant and proportional to the size of memory.

## 5.2.3. Implementation Constraints

In order to implement the memory allocation and garbage collection routines, two features are added to support object reallocation.

### 5.2.3.1. Protection Register

Every primitive FP function that is applied to an object, uses a set of registers to store information or pointers to different parts of that object. This information is essential for the correct execution of that primitive function. If we were to simply reallocate an object before a function has completed, the pointers in the registers would no longer point to the appropriate parts of the object, nor would they contain correct data. In Figure 10 the primitive function Transpose is applied to an input object. Pointer p is the current object pointer to a list of two lists, each consisting of two atoms. While performing the Transpose operation, registers ww, w and uu were used to store pointers to different parts of the partially transposed list object.

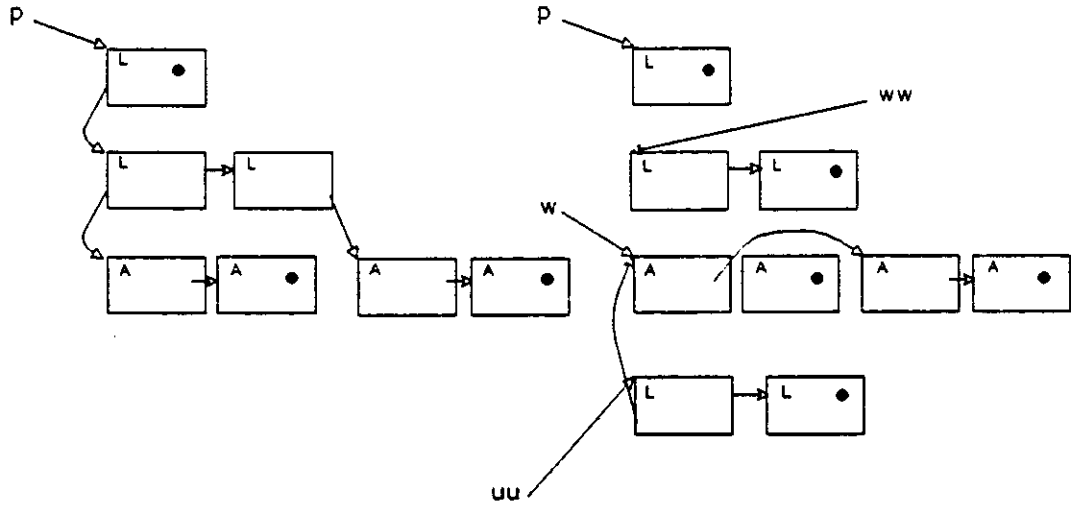


**Figure 10: A Partially Transposed Object**

If, during the execution of the primitive function, we were to reallocate the partially transposed object, we would also need to change the values of the used pointers. Since these pointers can point to any place in memory, and since, after reallocating the object, the relative positions of the used cells will change, one would have to repeat the interrupted function (now to a new object) rather than just continue its execution.

Let us consider another case, for example the Construct functional form of functions $f_1$ ... $f_n$ applied to an input object X. This is graphically represented in Figure 11. Let us also presume that the end of memory was reached while applying function $f_n$ to the $n^{th}$ copy of X. In this case, there are useful pointers ( in registers ) that belong not only to the interrupted function $f_n$ but to the construct functional form.

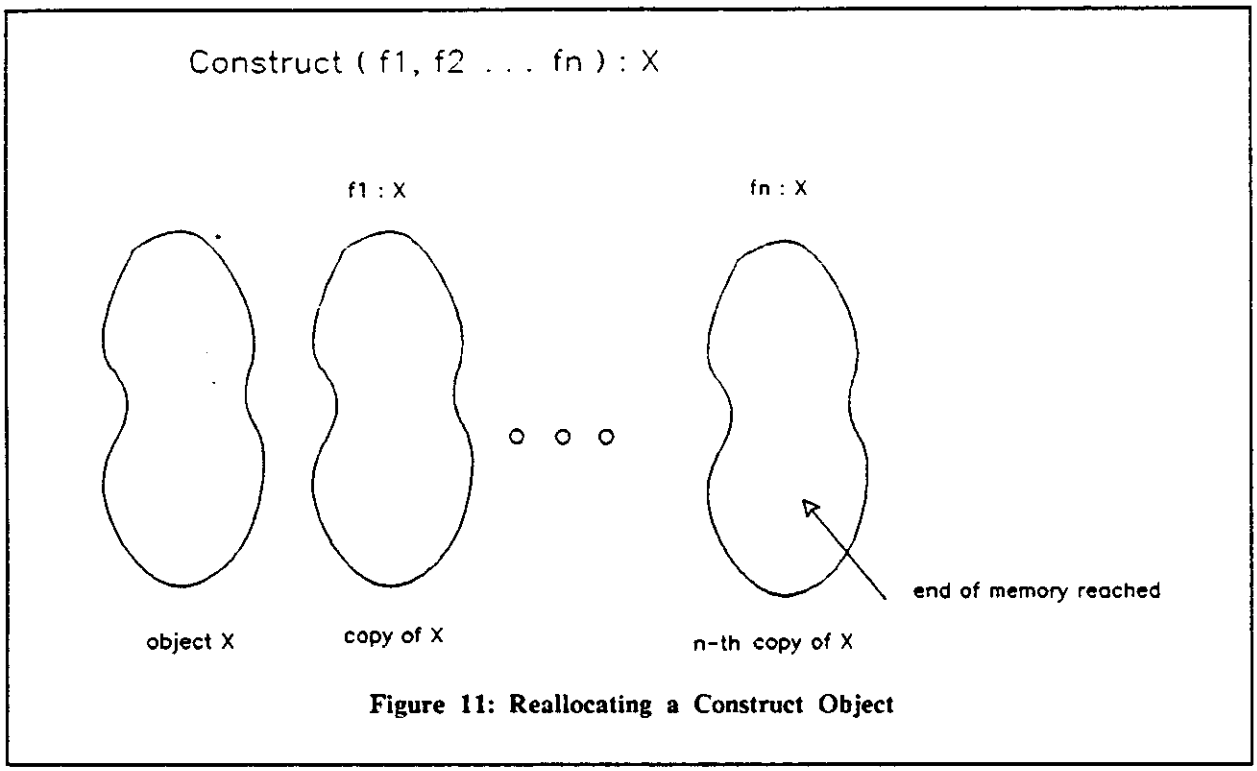


**Figure 11: Reallocating a Construct Object**

Therefore, we cannot simply reallocate the last function that was interrupted. Rather, we have to be able to reallocate the complete data structure formed after the completion of the construct. This means that we have to bring the construct functional form to a completion and then reallocate the complete object.

In order to correctly reallocate objects, reallocation is disabled during intervals determined by the interpreter. In the case of the construct example, reallocating objects would be disabled while executing the individual functions of the functional form.

To do this, a Protect register is used by the interpreter to indicate whether reallocation should be allowed. It is incremented by the interpreter at the beginning of the "protected" part of a function, and decremented after. The interpreter protection is performed mainly in those functional forms that use the Copy_Object routine to create a new object, in order to interpret another function.

When the end of memory is reached, the object will be reallocated only if the Protect register is equal to zero. A segment of the modified interpreter code, and a

sample FP program are given in figure 11.1 .

```
CASE CONSTRUCT
    BEGIN
       ProtectReg++;
         X = COPY_OBJECT(object_pointer);
         INTERPRET(X);
       ProtectReg--;
       IF ( !ProtectReg ) and ( MAXREACHED )
          pp = GARBAGE_COLLECTION(object_pointer);
    END;
 BREAK;

MAIN=TR @ { { DL,TR }, TL, SL2 } : X
```

Figure 11.1 Protected Portion Of Interpreter

In the above example of an FP program, only the Transpose that is outside the Construct functional form will lead to the reallocation of the object, if end of memory is reached. In order to be able to deal with the case when the interpreter reaches the end of memory and the Protect register is not equal to zero, we introduce a Scratchpad memory extension.

### 5.2.3.2. Scratchpad Memory Extension

In order to avoid the situation where the end of memory is reached but reallocation is disabled by the interpreter, a Scratchpad area is reserved at the end of memory. Whenever the memory allocator enters this area, garbage collection will be invoked as soon as the interpreter allows it. Therefore, the scratchpad area is meant only to allow the interpreter to bring the execution of the FP program to a point where the objects may be reallocated. The necessary size of the scratchpad, may vary depending on the nature of the programs used. In any case, it is a possibility that even the scratchpad area may not be sufficient for the interpreter to reach a safe point for reallocation purposes. A possible way to avoid such a situation would be to implement a "roll back" mechanism that would store the last function and object that had successfully executed. One could always restart from that point in the FP program, with the object reallocated to the beginning of memory. This mechanism was not implemented here. In Figure 12 the use of the scratchpad area is shown.

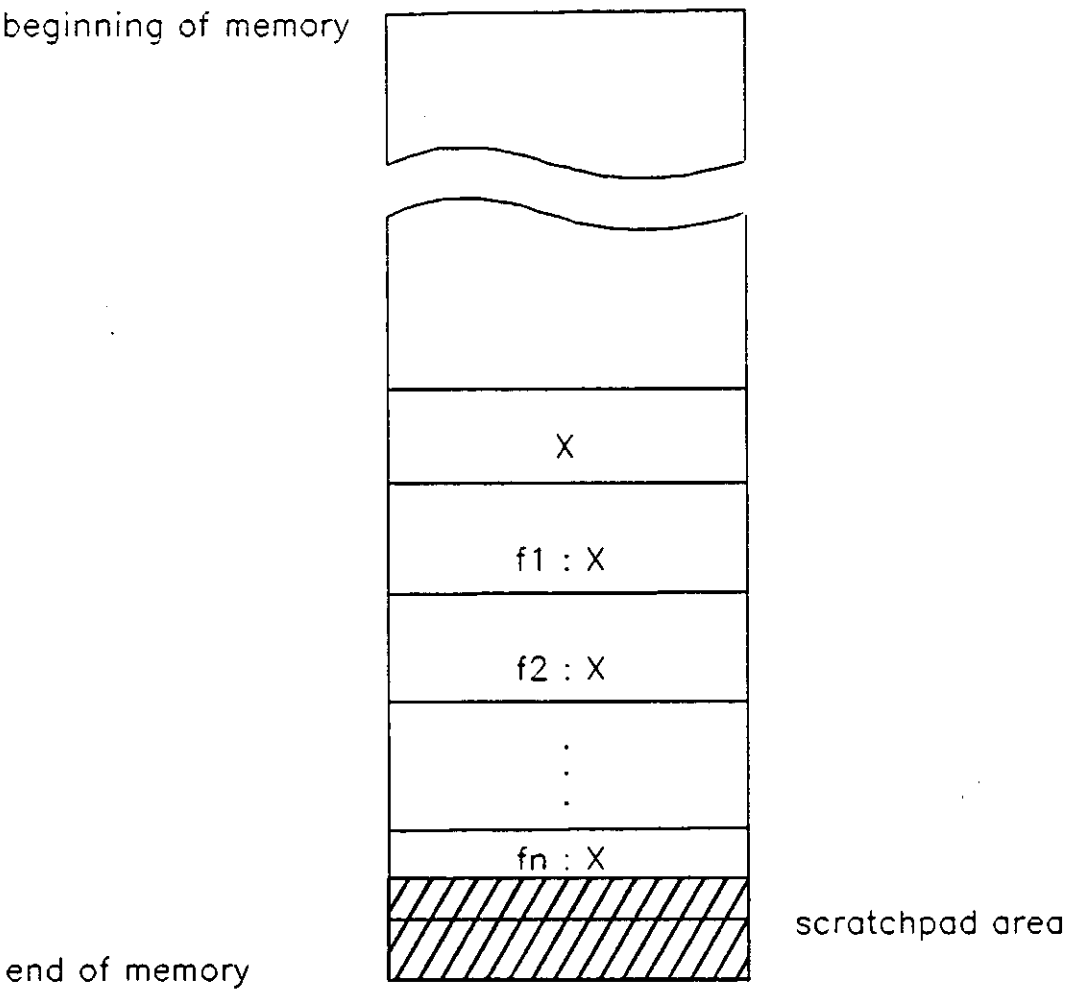


**Figure 12: The Scratchpad Memory Extension**

Figure 12 also shows how the construct functional form, applied to object X, is represented in memory. We can see that the function $f_n$ entered the scratchpad zone, and even though the end of memory was not reached, the garbage collection routine would be invoked.

We can note that in most cases not all of the scratchpad area is going to be used, so that in effect, not all of memory is used. The smaller the scratchpad is, the more chance there is that the interpreter will not be able to come to a point of safe

reallocation. On the other hand, by making the scratchpad larger, one uses the available memory less efficiently and eventually leads to more frequent garbage collection. Estimating an optimal scratchpad size was not considered here. Instead, a large enough scratchpad was used to execute the benchmarks.

By introducing the Scratchpad, the memory allocation algorithm is modified as given in figure 11.2.

```
ALLOCATE_SEQUENTIALLY()
BEGIN
   free_cell_pointer = free_cell_pointer + 1;
   IF ( free_cell_pointer = maxmem - scratchpad )
      maxreached = 1;
END;
```

Figure 11.2 Sequential Allocation with Scratchpad

### 5.2.3.3. Mark and CopyObject Stack

It should be noted that the Copy__Object and the Mark routines shown in figures 8.2 and 8.3 are recursive algorithms. This means that with each recursive call the return address and the variables have to be pushed onto a system stack. Such a stack could grow to a size larger than all of memory, even before the marking is finished.

An alternative non-recursive algorithm may use an explicit stack to save pointers to those cells that are being marked. In this case, no return address is pushed onto the stack so its growth will be slower than in the recursive algorithm. The idea here is that since we are reserving a stack only for marking, there is no need to store the return address to the routine. Therefore, because every time we mark a cell, we follow one of the two pointers ( if it is a list cell ) it is sufficient to store on the stack the address of the marked cell, so that one can then pop that address and apply the mark routine following the other cell pointer. If there are n cells in memory the maximum depth of the required stack is n. Even though the maximum required stack size is less than if the recursive algorithm were used, to reserve this much additional space for the stack is obviously uneconomical.

Therefore, the algorithm used here is a variant of the Schorr and Waite algorithm [Schor67] which uses a fixed size stack. If the stack overflows, a stack-less algorithm

is used to continue with the marking. The reason why the stack-less algorithm was not used in the first place is because it is very slow. It involves reversing successive links until a null pointer or an already marked cell is encountered. The reversed links are then used to restore the original data structure [Knuth73].

In the garbage collection algorithm considered here, the stack is allocated from the Scratchpad memory extension. The size of the stack depends on the size of the Scratchpad and on the amount of the Scratchpad used before the garbage collection routine was invoked.

It was mentioned earlier that we are not concerned with finding the optimal size of the scratchpad and that it is presumed to be large enough. This therefore means that we do not consider stack overflows. Such an assumption is perhaps too optimistic.

## 5.2.4. Performance Estimate

We can divide the overall time to perform memory allocation and garbage collection into three parts: the memory allocation time $T_a$, the memory overflow time $T_o$, and the overhead time necessary to support both memory allocation and garbage collection, $T_{oh}$. That is,

$$T = T_a + T_o + T_{oh} \tag{11}$$

### 5.2.4.1. Memory Allocation Time

The memory allocation time is given as:

$$T_a = N_a t_a \tag{12}$$

where $N_a$ is the number of cells allocated and $t_a$ is the time to allocate a single cell. The sequential allocation algorithm has the nice property that its allocation is simple and inexpensive in terms of the number of instructions executed per allocated cell. Since this is a frequent operation, it will have a major impact on the overall performance. Allocating a cell involves incrementing a pointer, testing for the end of memory, and branching. Whether the Scratchpad is used or not, does not affect the performance of the cell allocation, since in both cases, the value used for comparison can be stored in a register, rather than being computed.

If $K_a$ is the number of cycles it takes to allocate a single cell on a host machine, we can represent the allocation time in machine cycles as:

$$T_a = K_a N_a \tag{13}$$

### 5.2.4.2. Memory Overflow Time

Memory overflow time is the time taken to reallocate an object to the beginning of memory, and resume memory allocation. If $N_o$ is the number of overflows and $t_o$ is the time to resolve a single overflow, the complete overflow time is:

$$T_o = N_o t_o \tag{14}$$

We can divide this time into the three possible cases, mentioned earlier, and described in Figure 9 . In the first case, we can just reallocate the object to the beginning of memory; in the second case we need to search memory for a contiguous area of the correct size; if this space cannot be found, the object is reallocated in several stages. In each case, we have to mark the current object and determine which situation was encountered, ( see Garbage__Collection algorithm described in figure 8.4 ).

If $t_{o1}$ , $t_{o2}$ and $t_{o3}$ are the overflow times for the three cases, and if $F_1$, $F_2$ and $F_3$ are the frequencies with which they occur, we can write:

$$T_o = N_o(F_1 t_{o1} + F_2 t_{o2} + F_3 t_{o3}) \tag{15}$$

where

$$F_1 + F_2 + F_3 = 1 \tag{16}$$

In the first case, the time for marking and copying the object to the beginning of memory is proportional to the size of the object that is being copied. In the other two cases, besides marking and copying, one has to search memory for a contiguous area of adequate size.

If $S_o$ is the size of the object to reallocate, and M is the size of memory, we can express $t_{o1}$, $t_{o2}$ and $t_{o3}$ as:

$$t_{o1} = K_{o1}^{0} S_o \tag{17}$$

$$t_{o2} = K_{o2}^0 S_o + K_{o2}^1 M \tag{18}$$

$$t_{o3} = K_{o3}^0 S_o + K_{o3}^1 M \tag{19}$$

where the constants $K_{oi}^0$ and $K_{oi}^1$ represent the number of cycles it takes to execute the instructions on a host machine, for each case of i = 1,2 and 3.

We can therefore write:

$$T_o = N_o(F_1 K_{o1}^0 S_o + F_2(K_{o2}^0 S_o + K_{o2}^1 M) + F_3(K_{o3}^0 S_o + K_{o3}^1 M)) \tag{20}$$

or if we define the average overflow cost $K_{ov}$ as:

$$K_{ov} = (F_1 K_{o1}^0 S_o + F_2(K_{o2}^0 S_o + K_{o2}^1 M) + F_3(K_{o3}^0 S_o + K_{o3}^1 M)) \tag{21}$$

we can write:

$$T_o = N_o K_{ov} \tag{22}$$

### 5.2.4.3. Implementation Overhead Time

The implementation overhead in the sequential memory allocation and garbage collection algorithm consists of incrementing and decrementing the Protect register and testing whether the object may be reallocated or not. The testing for the end of memory is performed after every primitive function, but the interpreter protection is done only in certain functional forms.

Let $K_{pf}^0$ be the overhead of testing, and $K_{pf}^1$ the overhead of incrementing and decrementing the Protect register. Let $F_{oh}$ be the fraction of the overall number of executed functions $N_{pf}$ in which the interpreter protection overhead was encountered. We can then specify the total implementation overhead as:

$$T_{oh} = N_{pf} K_{pf}^0 + F_{oh} N_{pf} K_{pf}^1 \tag{23}$$

$$= N_{pf}(K_{pf}^0 + F_{oh} K_{pf}^1) \tag{24}$$

We can define the average overhead cost per allocated cell $K_{ohv}$ as:

$$K_{ohv} = (K^0_{pf} + F_{oh} K^1_{pf}) \tag{25}$$

and the overall overhead cost as:

$$T_{oh} = N_{pf} K_{ohv} \tag{26}$$

Consequently, the total time to perform the memory allocation and garbage collection algorithm is:

$$T_{seq} = N_a K_a + N_o K_{ov} + N_{pf} K_{ohv} \tag{27}$$

## 5.3. Linked List Memory Management

The Linked List Memory Management approach allocates cells from a Free List. When the end of memory is reached, the FP execution is interrupted, the useful data structures are traversed, marked, and the rest of the cells are linked back onto the Free List.

### 5.3.1. Linked List Memory Allocation

A new cell is allocated by accessing the top of a list of free cells. This means that at the beginning, memory has to be initialized and formed as an initial free list. In our first implementation of the linked list memory allocation algorithm, during the first pass through memory, cells were allocated sequentially. After the first garbage collection, a free list was formed, and new cells were then allocated from the free list. This had the nice property that it avoided memory initialization, and allocated fast in the initial pass. But, a significant drawback was that the time to allocate a single cell increased due to the checking whether we were in the first pass through memory or not. Since cell allocation is such a frequent operation, and since we are considering FP programs that will lead to garbage collection, it is better to keep the time of cell allocation as small as possible. The memory allocation algorithm implemented is shown in figure 11.3 .

```
ALLOCATE_LIST()
BEGIN
   free_cell_pointer = free_cell_pointer@.next;
   IF ( free_cell_pointer@.tag.mark_bit = 'S' )
      maxreached = 1;
END;
```

Figure 11.3 Linked List Allocation Algorithm

The linked list memory allocation policy is very similar to the sequential policy, with the only difference that the pool of available memory need not be contiguous. As far as the implementation overhead is concerned, the two approaches are identical, and they both use a Protect register and a Scratchpad area.

One should note that the Scratchpad memory extension is also implemented using a linked list data structure. In order to set the boundary for the beginning of the Scratchpad area, a bit is set in the tag of the cell located ( Maxmem-Scratchpad ) cells from the first cell in the free list. This bit is set while linking the free cells into one list.

## 5.3.2. Garbage Collection

When the memory allocator enters the scratchpad area, and when the interpreter allows garbage collection to be performed, the current object is marked, and the rest of memory is relinked into a single free list of cells. The Mark routine is the same as in the sequential approach and the Garbage__Collection routine is given in figure 11.4 .

```
GARBAGE_COLLECTION()
BEGIN
   MARK(current_object_pointer);
   cell_pointer1 = 1;
   WHILE ( cell_pointer1 < maxmem-1 )
      BEGIN
        cell_poiner1 = cell_pointer + 1;
        WHILE (cell_pointer1@.tag.mark= 'M'
                    AND cell_pointer1 < maxmem-1)
          BEGIN
            cell_pointer2 @.next = cell_pointer1;
            cell_pointer1 = cell_pointer1 + 1;
          END;
          cell_pointer2 = cell_pointer1;
      END;
END;
```

Figure 11.4 Linked List Garbage Collection Routine

One should note that the same stack problem found in the sequential approach is present here. Even though the relinking algorithm is not recursive, the marking routine is. In order to use part of the Scratchpad for storing the stack, a slight modification has to be made. While relinking the pointers in memory into a free list, the scratchpad cells are doubly linked. In this case a pointer to the top of the Scratchpad would in fact point to the beginning of the stack area in memory. The stack would not grow in the same way as in the sequential memory management approach but would follow the reversed pointers to consecutive stack locations.

Every time we want to save a value on the stack which is doubly linked, we first store the top__of__the__stack pointer, sp, in a temporary register, move sp to the next stack location using one of the pointers in the stack and then store the value in the stack at the position saved in the temporary register. Popping a value from the stack requires the reverse procedure. That is, sp is first saved, then the reverse pointer is followed to the next lower stack position, the stored value is popped and the forward pointer is replaced with the stored value of sp. The sp pointer thus always points to the next free location in the stack.

From the above stack description we can conclude that it is more expensive to perform marking in the linked list memory management scheme than in the sequential approach. Since there is no implicit ordering of consecutive stack locations, the two linked list pointers must always be preserved and maintained.

### 5.3.3. Performance Estimate

The performance estimate of the linked list memory allocation and garbage collection algorithm can be divided into the same three parts, as in the sequential approach. Therefore,

$$T = T_a + T_o + T_{oh} \tag{28}$$

The memory allocation time and the overflow time in the linked list algorithm will differ from the sequential approach. The implementation overhead is the same in both schemes because both algorithms use the Protect register and the Scratchpad memory extension in the same way.

#### 5.3.3.1. Memory Allocation Time

The memory allocation time may be specified in the same way as in the sequential approach, that is:

$$T_a = N_a t_a \tag{29}$$

The time $t_a$ to allocate a single cell in the linked list approach consists of accessing a cell in memory and moving a pointer to the cell-pointer register. Therefore, there are again three instructions: to move from memory, to test a tag in memory and to branch. If $K_a$ is the cost of a single allocation, the memory allocation time in machine cycles is:

$$T_a = K_a N_a \tag{30}$$

#### 5.3.3.2. Memory Overflow Time

Similarly to the sequential approach, we can write:

$$T_o = N_o t_o \tag{31}$$

The memory overflow time consists of the time to mark the current object and to relink the rest of memory. The first factor is proportional to the size of the object,

$S_o$, and the second to the size of memory M. Let $K_m$ be the cost of marking a single cell and $K_r$ the cost per cell of relinking the rest of memory. We can then write:

$$T_o = N_o(K_m S_o + K_r M) \tag{32}$$

This does not include the time to initialize memory. Even though it can not be considered as an overflow, the algorithm is the same, since memory is traversed and cells are linked together. We can easily add this time into the overflow factor, by adding 1 to the number of overflows. That is,

$$T_o = (N_o + 1)K_r M + N_o K_m S_o \tag{33}$$

We can define the average cost of overflow as:

$$K_{ov} = \frac{(N_o + 1)}{N_o} K_r M + K_m S_o \tag{34}$$

and we can therefore write the expression for the overflow time as:

$$T_o = N_o K_{ov} \tag{35}$$

The linked list approach to garbage collection avoids the main drawback of the sequential policy, that is, there is no need to determine whether a certain object may be copied to another location or not. There is no constraint made on memory to be contiguous in order to allocate cells.

Knowing that the implementation overhead in the interpreter, is identical to the one found in the sequential approach, we can write the overall performance of the linked list memory allocation and garbage collection policy:

$$T_{list} = N_a K_a + N_o K_{ov} + N_{pf} K_{ohv} \tag{36}$$

## 5.4. Stack Memory Management

The Stack Memory Management algorithm is based on the fact that cells discarded, after a primitive function is applied to an object, are still connected by pointers used in the original data structure. If we save pointers to these data structures during the execution of the primitive functions, we perform automatic, that is dynamic, garbage collection. New cells may then be allocated either from the stored garbage data structures or from the yet unaccessed parts of memory.

### 5.4.1. Garbage Data Structures

In order to store pointers to the garbage data structures left behind after a function is applied to an object, the FP functions had to be modified to return two pointers: a pointer p to the newly formed object, and a pointer g to the garbage data structure. A stack is used to store all the garbage pointers. In most primitive functions, no other modification was necessary other than pushing the garbage pointer. In order to understand what modifications might be necessary, let us look at an example shown in Figure 13. The SEL primitive function is applied here to a list object. The current object pointer p points to the list ( A, B ... , N ). The returned pointer p points to the selected element.

In order to obtain a single garbage data structure pointed to by pointer g, one extra instruction had to be added to the SEL primitive function. That is, after selecting the appropriate element, the next field of the previous element in the list is modified to point to the element following the selected object.

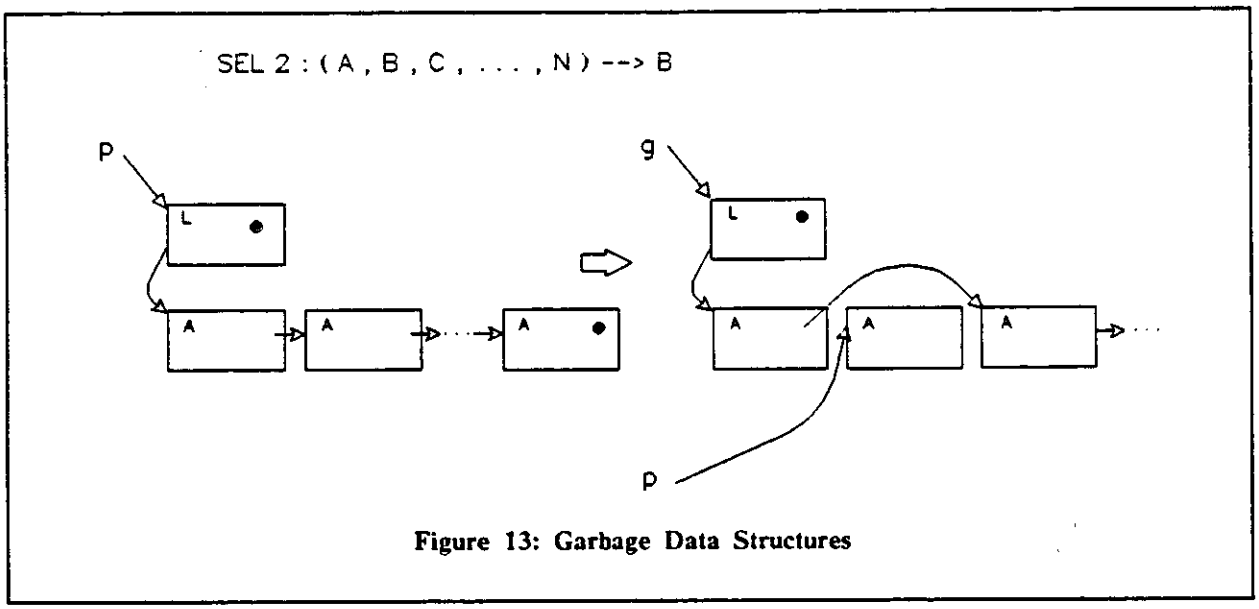


**Figure 13: Garbage Data Structures**

In some primitive functions, the modification consists of merely terminating a garbage data structure with a NULL pointer. We can also note that not every primitive function produces garbage. For example, functions like DISTL and DISTR only cause the object to grow, without leaving any garbage. Some functions, like the LOGICAL or ARITHMETIC ones, produce garbage pointers to only 2 cells, and some like SELECT produce garbage of variable size.

## 5.4.2. Stack Memory Allocation

Once there are pointers pushed onto the stack, new cells may be allocated by traversing the garbage data structures. There are several ways to allocate cells using such a stack. One way would be to traverse the data structures, either breadth or width first, and allocate the end cells of the tree-like garbage structures. In this way,

the size of the stack would only decrease when the data structure, pointed to by the pointer on the stack, is fully allocated. Meanwhile, new pointers may be pushed onto the stack. In this case, we would traverse the data structures for every cell that we need to allocate. Since we know that this is the most frequent operation, we would like to minimize its cost. Therefore, traversing does not seem the most efficient thing to do.

Rather than doing this, in the algorithm implemented, a new cell is allocated by popping the top pointer of the stack. If the allocated cell is a list, then its two pointers ( first and next element ), have to be pushed onto the stack again. If the popped pointer points to an atom, then only one pointer is pushed onto the stack. If the allocated cell is a null list, or an atom that does not have a next element, nothing is pushed onto the stack.

Therefore, every time we allocate a list cell, the size of the stack will increase by one. Every time we allocate an atom it will not change in size, and if we allocate a null list or an atom that does not have a next element, the size of the stack will be reduced by one. We are therefore concerned that the size of the stack be within reasonable bounds to be actually implemented.

In the allocation algorithm given in figure 13.1, memory is allocated sequentially in the first pass because sequential allocation is less costly than allocating from a stack. Once the end of memory is reached for the first time, pointers to new cells are allocated from the stack.

```
ALLOCATE_STACK()
BEGIN
 IF ( free_cell_pointer < maxmem-1 ) and ( firstpass )
    free_cell_pointer = free_cell_pointer + 1
 ELSE BEGIN
    firstpass = 0;
    free_cell_pointer = TOP_OF_STACK();
    END;
END;
```

Figure 13.1 Stack Memory Allocation

Figure 13.2 depicts the Top__Of__Stack routine which pops the top of the stack and test whether the cell about to be allocated is a list or an atom cell. The appropriate pointers are then pushed onto the stack.

```
TOP_OF_STACK();
BEGIN
 pointer = POP();
 IF ( pointer@.tag.mark_bit = LIST )
    IF (pointer@.value NOT = NULLL )
       PUSH(pointer@.value);
 IF (pointer@.next NOT = NULLL)
    PUSH(pointer@.next);
 RETURN(pointer);
END;
```

Figure 13.2 Maintaining Garbage Data Structures

### 5.4.2.1. Stack Size

As it was mentioned earlier in this section, we are concerned with the size of stack required to support this memory management policy. Using the memory management simulator, two benchamrks were executed. The Matrix Multiplication benchmark multiplying matrices of size 5x5 several times ( 4 or 5 times ) and the Quicksort benchmark sorting lists of size 8 to 10. In each case memory size was fixed at 1000 cells. Both benchmarks required several times the available number of cells, so that they had to perform garbage collection. In Figure 14 on page 61, we show the stack histogram of the Matrix Multiplication benchmark, for a memory size of 1000 cells. A similar histogram was obtained for the Quicksort benchmark.

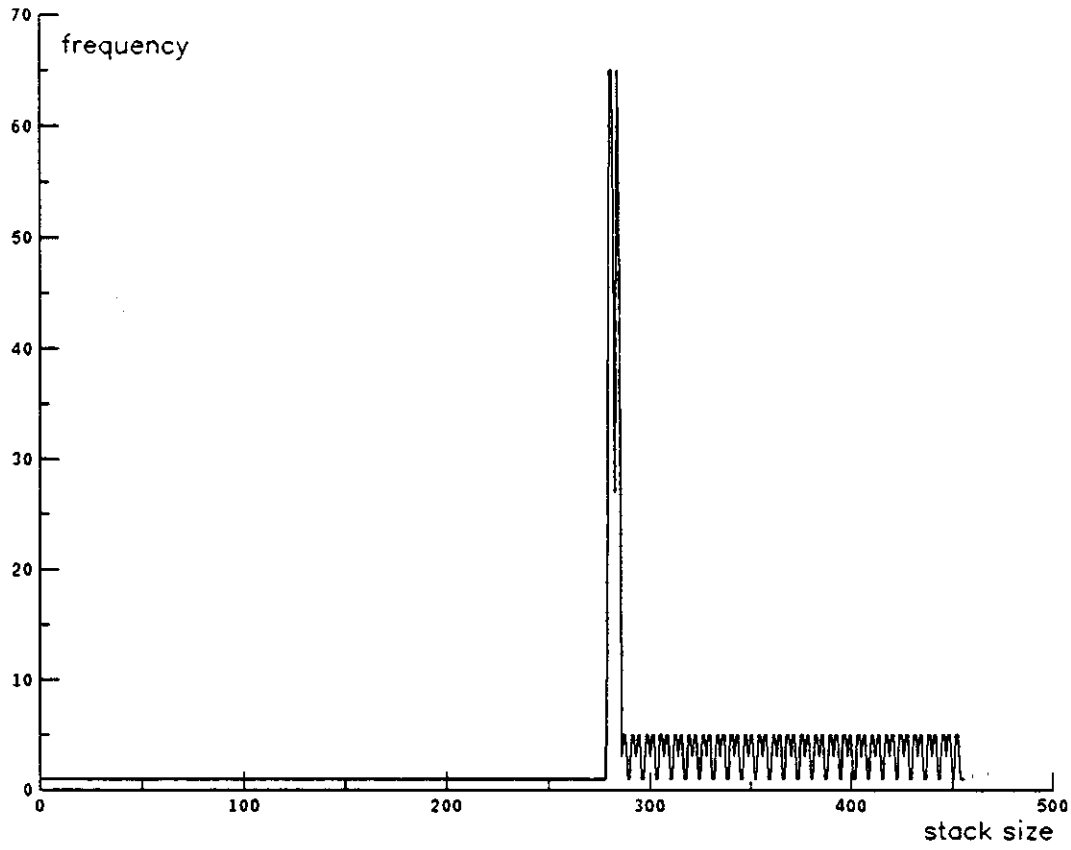


**Figure 14: Stack histogram for MM benchmark**

From the stack histogram for Matrix Multiplication we can note two features. First, we see that the stack size has grown to a size comparable to that of memory. Second, we note that more than half the stack is accessed only once.

By looking at the histogram, we can conclude that during the first pass through memory, all the garbage pointers were just pushed onto the stack, without being used at all. This would account for the large portion of the histogram where the stack location is accessed only once, to push a pointer. The larger the memory, the larger this portion of the stack would be. In order to avoid this undesirable effect, we use an Immediate Stack Allocation approach.

## 5.4.3. Immediate Stack Allocation

In the Immediate Stack Allocation algorithm, new cells are allocated from the stack, as soon as there are pointers available. Only when the stack is empty will a cell be allocated from a sequential pool of available cells. The new stack allocation algorithm is shown in figure 14.1

```
ALLOCATE_STACK_IMMEDIATE()
BEGIN
  IF ( SP > 0 ) free_cell_pointer = TRAVERSE()
  ELSE free_cell_pointer = memtop + 1;
END;
 Figure 14.1 Immediate Stack Allocation Algorithm
```

The histogram obtained in this case for the same Matrix Multiplication benchmark is shown in the Figure 15 on page 63.

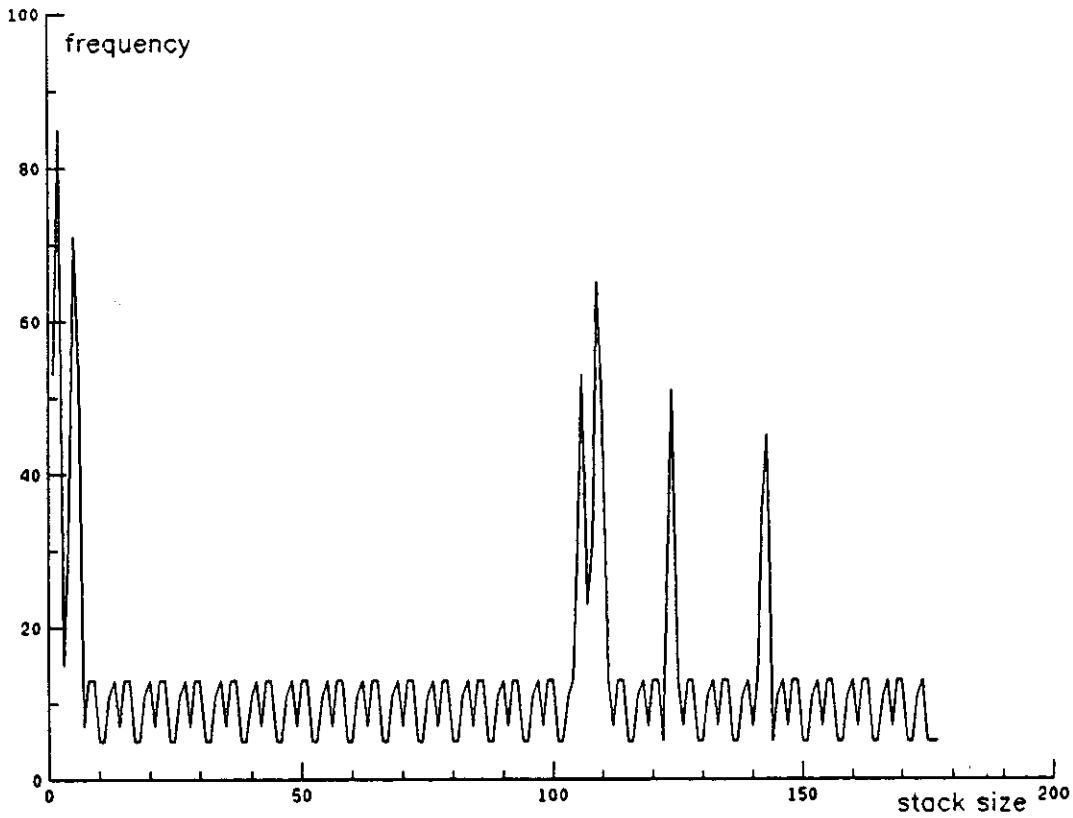


**Figure 15: Stack Histogram for MM and Immediate Garbage Collection**

From the histogram we can see that with the immediate approach to garbage collection, we have eliminated that portion of the stack that was accessed only once. Also, the stack size is reduced to less than half, with a more efficient use of the stack.

## 5.4.4. Special Purpose Garbage Registers

Compared to the first approach, the immediate garbage collection algorithm reduced the size of the stack used, even though it remained significant. In order to find alternative ways of reducing the growth of the garbage collection stack, let us look more closely at the Matrix Multiplication benchmark. We can note that it performs $2n^3 - n^2$ multiplications or additions. For matrices of size n = 5, that would

account for 255 times that a pointer is pushed. If we know that the garbage produced after each arithmetic operation consists of only two cell, then we are saving 255 garbage pointers in order to collect 500 cells. This is obviously a disastrous proportion.

In Figure 16 we show how the Logic and Arithmetic primitive operations are performed on list objects that contain 2 atoms. The operator * used in the diagram represents any binary operator.

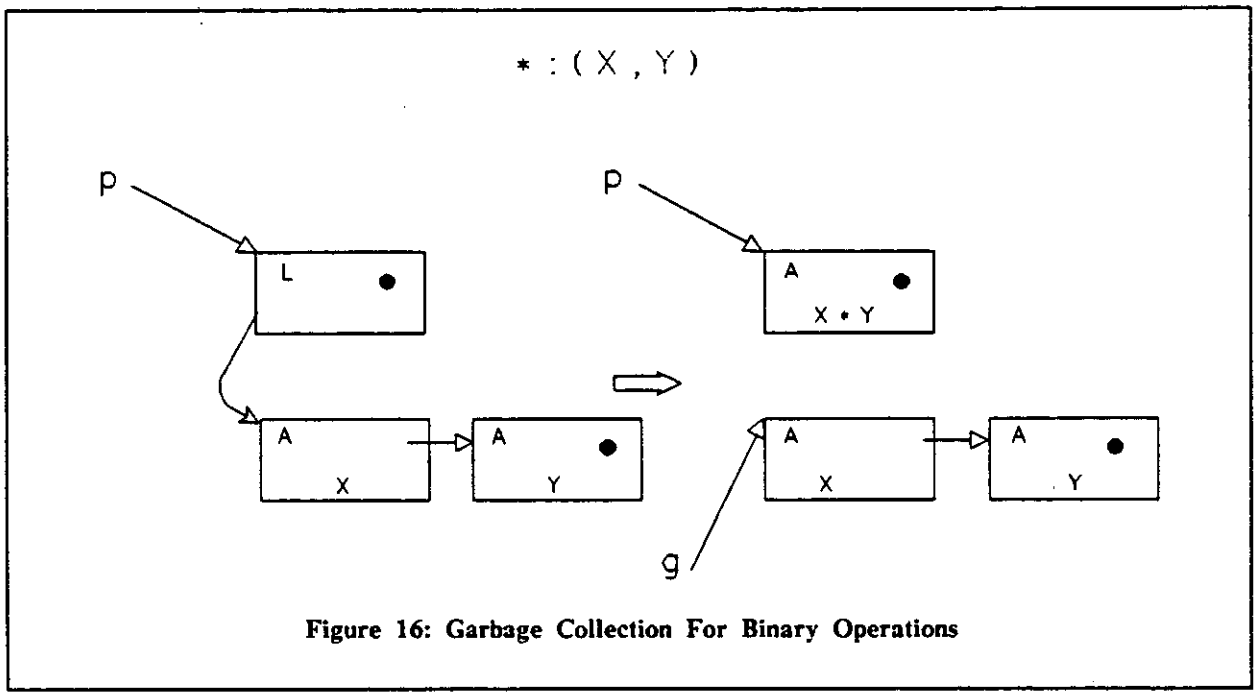


**Figure 16: Garbage Collection For Binary Operations**

Applying a binary operation * to a list object of two elements will produce a two cell garbage data structure pointed by pointer g. One of the cells is reused to store the result and form the output object pointed to by pointer p. We could choose in our implementation to reuse any of the available cells for the result, and return a garbage pointer to the rest of the object. By selecting to reuse the list cell, we return a garbage pointer to two atoms. This is useful since when these garbage cells are reallocated, only one pointer will be returned to the stack as opposed to two pointers for list garbage cells. This is important since we mentioned earlier that popping an atom cell from the stack will actually reduce the stack size by one.

What is more important to note from the binary operation garbage data structure is that we know the exact size and structure of the garbage produced. This means that we could, instead of pushing a pointer to this data structure, just add it to a reserved data structure.

One way to implement this is to have two specialized stack locations, or registers, that will be used for strictly binary operators. One register will contain a pointer to the beginning of the garbage data structure, and the other register will point to the end. In this case, each binary operator would just string its two atoms onto an already existing data structure. Therefore, the 255 pointers, pushed in the Matrix Multiplication benchmark, would no longer be pushed, and would not contribute to the stack growth.

Therefore, in all of the binary operations used in the FP programs, a garbage data structure will just use two reserved stack locations, stack(1) and stack(2). After modifying the execution of the binary FP primitive functions, the histogram obtained for the same Matrix Multiplication benchmark is given in Figure 17 on page 66.

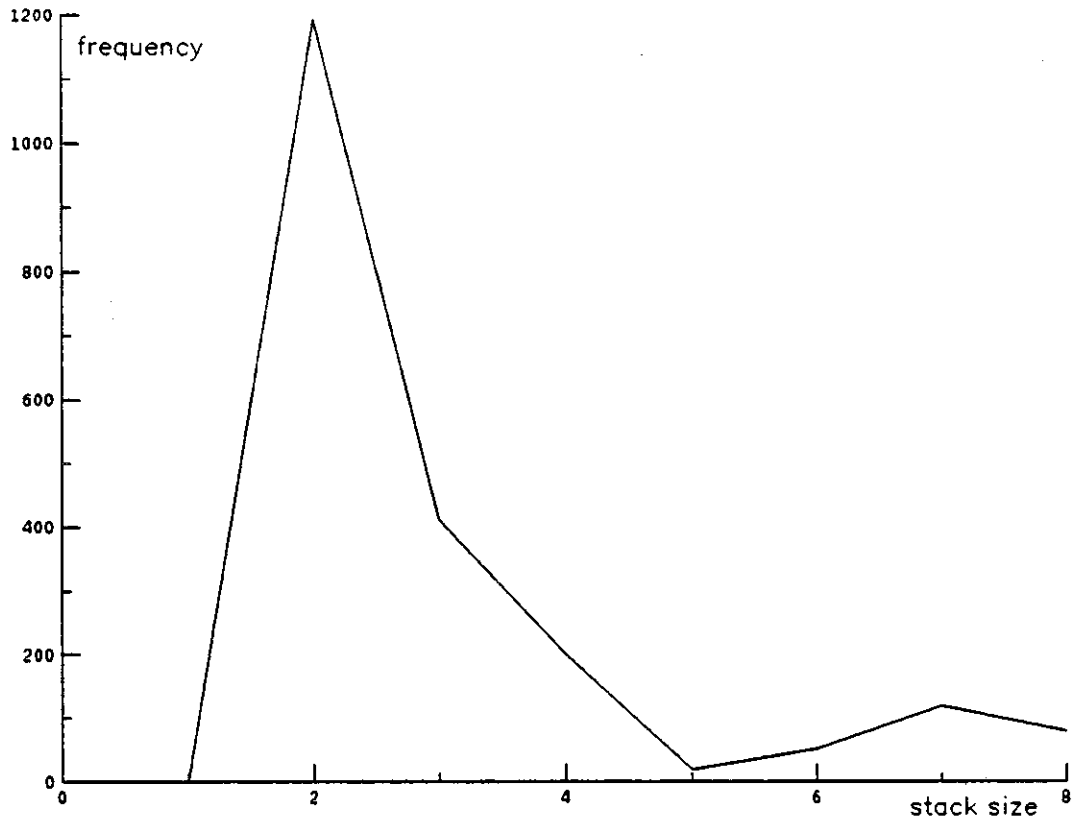


**Figure 17: Histogram for Immediate Allocation and Registers, MM**

We can see a significant improvement. Only eight stack locations were used, two of them, stack(1) and stack(2), dedicated to binary operations. The stack location that was used to string the atom cell from the binary operations, stack(2), was accessed 1200 times ! The improvement is perhaps significant because Matrix Multiplication is so arithmetic intensive. In order to see whether the algorithm is effective for other benchmarks, we used Quicksort which has very few binary operations. The stack histogram in this case is shown in the Figure 18 on page 67. Here again we see a significant improvement in the size of the stack used, and in the use of stack locations.

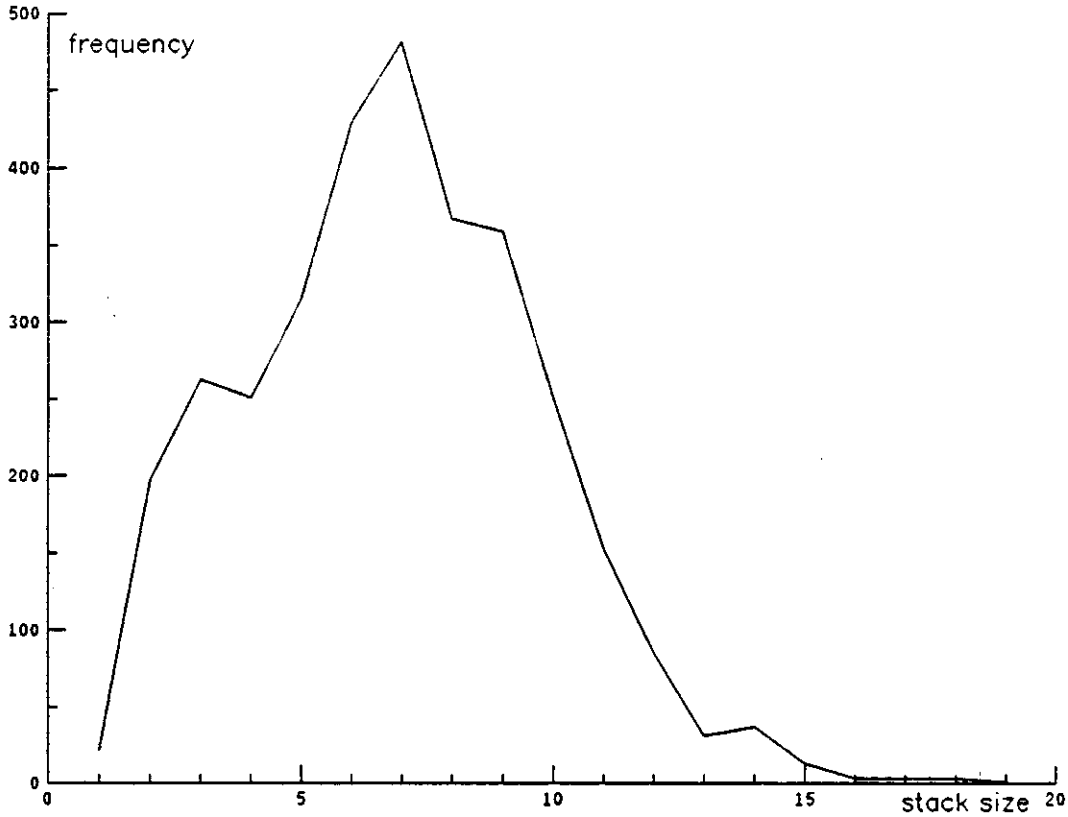


**Figure 18: Histogram for Immediate Allocation and Registers, Quicksort**

The same logic applied to the binary operations, may be applied to any primitive operation whose garbage data structure is predictable. This is the case for the TRANSPOSE, APPENDL, APPENDR, CONCATENATE and a few other functions. In Figure 19 on page 68 we show how the APPENDL and CONCATENATE primitive functions always leave behind a garbage data structure known in advance. Modifying these primitive functions accordingly contributes to a very efficient use of a small stack structure for garbage collection.

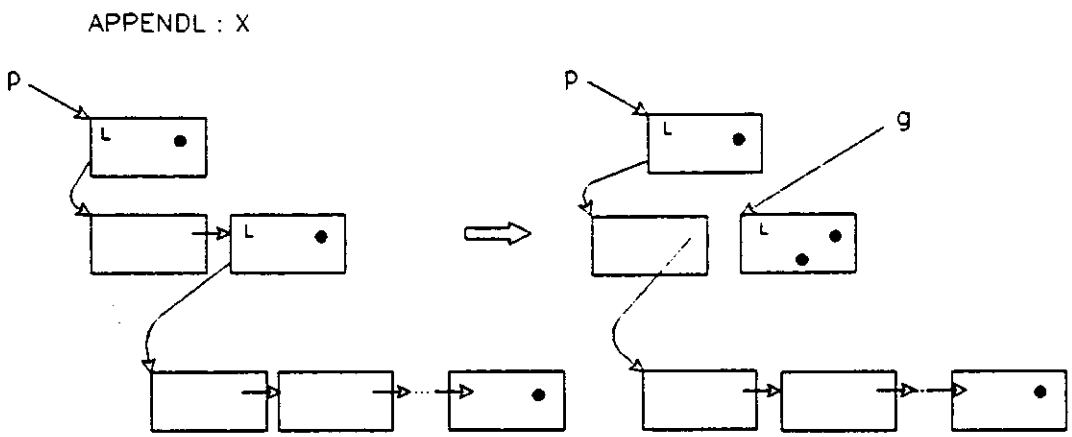


**Figure 19: APPENDL Garbage Data Structure**

From the figure above depicting the AppendL garbage data structure, we can see that the garbage is only one cell with two NULL pointers. Allocating this garbage cell from the stack does not require any further storing of pointers back onto the stack. It therefore reduces the size of the stack. In general it seems like a useful policy to form as many garbage cells will NULL pointers.

It is interesting to note that in the stack allocation algorithm, the amount of memory used increases only if the current object grows beyond the number of physically allocated cells. This is a direct consequence of the fact that garbage cells are reused as soon as they are available. In the case of the Matrix Multiplication benchmark that uses 2752 cells and an available memory size of 1000, only 337 cells were physically allocated. For the Quicksort benchmark, 1686 cells were allocated from a physical memory size of only 223 cells. One can therefore note a high level of locality of program execution in memory, which may be especially interesting in a multiprocessor or a multiprogramming environment. This useful property will be discussed later in this chapter.

## 5.4.5. Performance Estimate

In the stack memory allocation and garbage collection approach, the time performance is divided into two parts: the time to allocate cells $T_a$ and the implementation overhead time $T_{oh}$.

$$T = T_a + T_{oh} \tag{37}$$

We can note that there is no overflow time penalty here. In fact, overflow can occur here only if memory is literally overflowed with a single object. Such a condition is not detected in any of the implementations considered.

### 5.4.5.1. Stack Allocation Time

The allocation time is again given as:

$$T_a = N_a t_a \tag{38}$$

The time to allocate a single cell, $t_a$, varies depending on how the allocation algorithm is coded. From the Immediate Stack Allocation algorithm shown in figure 13.1, we can note that for each allocation one needs to pop the pointer from the stack, access the memory cell it points to, test whether it is an atom or a list cell, and then store the appropriate pointers onto the stack. The allocation time also depends on whether the allocated cell is an atom, a list cell or whether the stack was empty and the new cell was allocated sequentially. In case it is an atom, only one pointer is pushed onto the stack; if it is a list cell, two pointers are accessed and, if they are not NULL pointers, pushed onto the stack; if the stack is empty, a cell pointer is just incremented.

If $K_{av}$ is the average cost of allocating a single cell, we can write:

$$T_a = N_a K_{av} \tag{39}$$

If $F_{aa}$ is the frequency with which a stack pointer points to an atom, $F_{al}$ the frequency with which a list cell is referenced and $F_{as}$ the frequency of allocating cells by incrementing a pointer when the stack is empty, and if $K_{aa}$, $K_{al}$ and $K_{as}$ are the implementation dependent costs of each allocation, we can write:

$$K_{av} = F_{aa}K_{aa} + F_{al}K_{al} + F_{as}K_{as} \tag{40}$$

That is,

$$T_a = N_a(F_{aa}K_{aa} + F_{al}K_{al} + F_{as}K_{as}) \tag{41}$$

### 5.4.5.2. Implementation Overhead Time

The implementation overhead in the stack algorithm consists of pushing a pointer onto a stack, and modifying some primitive functions so that the pointer points to the correct garbage data structure. Some primitive functions that do not create garbage do not produce any overhead. Others like the TAIL or FRONT require an extra instruction to properly terminate the garbage data structure with a NULL pointer, ( see Figure 13 on page 58 showing garbage data structures ). Let $K_{ohv}$ be the average cost of overhead per executed primitive function. We can then write:

$$T_{oh} = N_{pf}K_{ohv} \tag{42}$$

To find the average overhead per primitive function, we have to sum all the individual overheads multiplied by their frequency. If $K_{ohpf}$ is the overhead found in each primitive function and $F_{pf}$ is the frequency of each function, we can write:

$$K_{ohv} = \sum_{pf} K_{ohpf}F_{pf} \tag{43}$$

We can now write the overall time estimate for the stack memory allocation and garbage collection algorithm to be:

$$T_{stack} = N_aK_{av} + N_{pf}K_{ohv} \tag{44}$$

## 5.5. Performance Comparisons

In order to compare the three approaches to memory management, we will express the time performances as a function of the number of allocated cells. If we normalize the expressions to the size of memory M used, we can evaluate the time performance per memory cell versus the number of allocations per physical cell of memory. If we divide all three equations by the size of memory, we can write:

$$\frac{T_{seq}}{M} = \frac{N_a}{M} K_a^s + \frac{N_o}{M} K_{ov}^s + \frac{N_{pf}}{M} K_{ohv}^s \tag{45}$$

$$\frac{T_{list}}{M} = \frac{N_a}{M} K_a^l + \frac{N_o}{M} K_{ov}^l + \frac{N_{pf}}{M} K_{ohv}^l \tag{46}$$

$$\frac{T_{stack}}{M} = \frac{N_a}{M} K_{av}^{st} + \frac{N_{pf}}{M} K_{ohv}^{st} \tag{47}$$

where $N_a$ is the number of allocated cells, $K_a$ is the cost of allocating a cell, $K_{av}$ is the average cost of allocating cells, $K_{ov}^l$ is the average cost of overflow in the list approach, $K_{ov}^s$ is the average cost of overflow for sequential policy, $K_{ohv}^{st}$ is the average implementation overhead per primitive function and $K_{ohv}^{s,l}$ the average implementation overhead per allocated cell.

## 5.5.1. Number of Overflows $N_o$

Memory overflow is considered only in the two static approaches to garbage collection, that is, in the sequential and linked list memory management policy. An overflow situation will occur whenever all of memory is allocated. The number of overflows is inversely proportional to the size of memory, and directly proportional to the number of allocated cells.

In the linked list approach, the ratio of the number of allocated cells and the size of memory, will exactly indicate the number of overflows. In the sequential approach, there is extra cell allocation due to the copying of the object that leads to overflow. Therefore, the number of allocated cells in the sequential policy will be greater than the number obtained in the linked list or stack approach. Let $N_a^l$ be the number of allocated cells in either the stack or linked list policy. We can specify the number of overflows in the linked list approach using the floor function $[x]$

$$N_o^l = \left[\frac{N_a^l}{M}\right] \tag{48}$$

That is, because there is no extra copying of objects in memory, overflow will occur every time M number of cells are allocated.

Let $S_o$ be the average size of the object copied to the beginning of memory in the sequential approach. That means that every time an overflow occurs, we start reallocating from a point $S_o$ cells from the beginning of memory. Therefore, in the average, overflow will occur after $(M - S_o)$ cells are allocated in the linked list or stack approach. We can now write the number of overflows in the sequential approach $N_o^s$ as:

$$N_o^s = \left[\frac{N_a^l}{(M - S_o)}\right] \tag{49}$$

The number of allocated cells in the sequential algorithm depends on the number of copying of objects due to overflow. We can write,

$$N_a^s = N_a^l + N_{ac}^s \tag{50}$$

where $N_{ac}^{s}$ is the number of extra allocations performed in the sequential approach. In the case where we directly copy an object to the beginning of memory, the number of extra allocations is equal to the size of the object. If the object is first copied to a temporary location and then to the beginning of memory, the number of allocations is double the size of the object. In the third case, where we reallocate the object in several stages, the number of extra allocations depends on the number of times n the object is copied first to different areas in memory, before it is finally reallocate to the beginning of memory.

Depending on the frequency with which each case occurs, we can compute the average number of extra cell allocations per overflow, that is:

$$N_{acv} = F_1 S_o + 2F_2 S_o + nF_3 S_o \tag{51}$$

Therefore the total number of extra cell allocations is :

$$N_{ac} = N_o N_{acv} \tag{52}$$

In all the benchmarks performed, a count was kept of the number of functions executed and the number of cells allocated. Let $N_{apf}$ be the average number of cells allocated per executed primitive function, that is:

$$N_{apf} = \frac{N_a}{N_{pf}} \tag{53}$$

Using this expression and the expression for the number of overflows, the three memory management performance estimates may be written as:

$$\frac{T_{seq}}{M} = (N_a^l + N_o N_{acv}) \frac{K_a^s}{M} + \frac{N_o}{M} K_{ov}^s + \frac{N_a^l}{N_{apf}^l} \frac{K_{ohv}^s}{M} \tag{54}$$

$$= \frac{N_a^l}{M}(K_a^s + \frac{K_{ohv}^s}{N_{apf}^l}) + \left[\frac{N_a^l}{(M - S_o)}\right](N_{acv} K_a^s + K_{ov}^s) \frac{1}{M} \tag{55}$$

$$\frac{T_{list}}{M} = \frac{N_a^l}{M} K_a^l + \left[\frac{N_a^l}{M}\right] \frac{K_{ov}^l}{M} + \frac{N_a^l}{N_{apf}^l} \frac{K_{ohv}^l}{M} \tag{56}$$

$$= \frac{N_a^l}{M}(K_a^l + \frac{K_{ohv}^l}{N_{apf}^l}) + \left[\frac{N_a^l}{M}\right] \frac{K_{ohv}^l}{M} \tag{57}$$

$$\frac{T_{stack}}{M} = \frac{N_a^l}{M} K_{av}^{st} + \frac{N_a^l}{M} \frac{K_{ohv}^{st}}{N_{apf}^{st}} = \frac{N_a^l}{M}(K_{av}^{st} + \frac{K_{ohv}^{st}}{N_{apf}^{st}}) \tag{58}$$

The implementation costs depend on which host machine one implements the storage management policies. The three approaches are compared for a Motorola 68000 microprocessor.

## 5.5.2. A Motorola 68000 Implementation

The memory management routines for all three approaches were hand compiled for the Motorola 68000 microprocessor, [Motor82]. The cost of each routine is computed by adding the number of cycles each instruction takes to execute. Tables showing the different instruction execution times obtained from the Motorola 68000 Microprocessor User's Manual are shown in Appendix C.

Since some of the performance parameters are program dependent, the following results are based on the Matrix Multiplication and Quicksort benchmarks.

### 5.5.2.1. Memory Allocation Cost

Allocating a cell in the sequential approach requires a cell-pointer to be incremented, end of memory to be tested to see whether the scratchpad memory was entered, and a branch to be taken, (see figure 8.1). This accounts for 40 cycles, that is,

$$K_a^s = 40 \tag{59}$$

In the linked list policy, allocating a cell requires moving a pointer to the cell-pointer register and testing whether the scratchpad memory was entered. The cost of these three instructions is 34 cycles. It might be surprising that the cost is less than that obtained in the sequential approach. This is because incrementing a pointer takes 16 cycles, whereas moving a pointer from memory takes 12. Also, in the linked list

approach, end of memory is tested by comparing the byte tag with an immediate value, which takes 12 cycles. In the sequential policy, the comparison takes 14 cycles. Therefore,

$$K_a^l = 34 \tag{60}$$

In the stack approach, the cost of allocation depends on how the allocation routine is coded. One possibility is to have the cell, pointed to by the cell-pointer, immediately moved into registers. All testing for NULL pointers is then done within the Motorola 68000 processor, where all the MOV instructions take only 4 cycles. The costs of allocating an atom, a list cell or a cell by just incrementing a pointer, are:

$$K_{aa}^1 = 82 \quad K_{al}^1 = 86 \quad K_{as}^1 = 34 \tag{61}$$

If instead of this approach, we test whether the cell pointers are equal to NULL while the cell is in memory, we avoid moving one pointer if the cell is an atom. Also, if both pointers are NULL pointers, none are pushed onto the stack. In this case the costs obtained are:

$$K_{aa}^2 = 74 \quad K_{al}^2 = 116 \quad K_{as}^2 = 34 \tag{62}$$

We can see that in the second case allocating an atom cell is less costly but allocating a list cell is significantly more expensive. Which implementation one should use depends on the ratio of the number of allocated list cells to the number of allocated atom cells. For the two cases to have an equal average cost of allocation, the ratio would have to be:

$$\frac{F_{al}}{F_{aa}} = \frac{(K_{aa}^1 - K_{aa}^2)}{(K_{al}^2 - K_{al}^1)} = \frac{(82 - 74)}{(116 - 86)} = 0.26 \tag{63}$$

Therefore, if there are at least 4 times as many atoms allocated than list cells, then the second case will give a lower average cost per allocated cell. Otherwise the first case is more efficient. The following distribution was observed for the Matrix Multiplication and Quicksort benchmarks.

$$F_{aa}^{MM} = 0.65 \quad F_{al}^{MM} = 0.2 \quad F_{as}^{MM} = 0.15 \tag{64}$$

$$F_{aa}^{Q} = 0.55 \quad F_{al}^{Q} = 0.3 \quad F_{as}^{Q} = 0.15 \tag{65}$$

In both cases, there was a higher number of allocated atoms, but not 4 times the number of list cells. Therefore, the first case is considered here. Using the above frequencies, the average cost of cell allocation can be found to be 75.6 cycles for Matrix Multiplication and 75.9 for Quicksort.

$$K_{av}^{st,MM} = 75.6 \qquad K_{av}^{st,Q} = 75.9 \tag{66}$$

In order to improve the cell allocation cost for the stack allocation algorithm, a certain property of the Motorola 68000 may be used. That is, during a MOV instruction, among the condition codes that are set is the Negative flag. If we were to chose to place the bit that will be used to distinguish between atom and list cells in the most significant position of the long word ( 32 bits ), then no extra testing is necessary. One can immediately after a MOV instruction BRANCH on a condition code for negative or positive numbers. The Motorola stores integers in two's complement so that the most significant bit is a sign indicator as well.

Avoiding a branch instruction would save in both cases, that is, if an atom or a list cell is allocated. From the instruction performance tables shown in Appendix C, we can see that the COMP.byte reg,reg instruction takes 4 cycles to execute. This would reduce the atom allocation time to $K_{aa}^{1} = 78$ and $K_{al}^{1} = 82$ . Using the same frequencies as before, the average cell allocation time for atoms and lists in both benchmark cases is

$$K_{av}^{st,MM} = 70.5 \qquad K_{av}^{st,Q} = 71.6 \tag{67}$$

### 5.5.2.2. Overflow Cost

In the sequential memory management policy, with every overflow the current object has to be marked, and depending on its distribution, one of the three cases, mentioned earlier, will occur. In any case, the object will be marked and copied. In order to compute the cost per cell of marking and copying, the MARK and COPY_OBJECT routines are hand assembled into Motorola 68000 code. We can note that both routines are recursive, and that they both have one pointer as a parameter.

If an explicit stack approach ( see section 5.2.3.3 ) is used, one can avoid the overhead of storing the mark routines return address, see figure 8.3 . If the cell pointer ( passed as a parameter ) is not a NULL pointer, the cell it points to is marked and a count register (a global variable) representing the size of the marked object, is incremented. A flag is set indicating whether the count value is less than the pointer value. This flag is used to determine whether one can reallocate the object to the beginning of memory without writing over cells that belong to the object. Marking then continues depending on whether the marked cell was an atom or a list cell. The Mark routine will complete marking a single cell by popping the register values and the address of the next cell to mark. If the explicit stack is empty, the marking is completed. This accounts for an overall of 196 cycles per marked cell.

The Copy__Object routine ( figure 8.3 ) also requires pushing a pointer and local variables onto a stack. It also calls the Allocate__Cell routine which returns a pointer to a new cell. All these instructions add up to 356 cycles per copied cell. That is,

$$K_{o1}^{0} = 196 + 356 = 552 \tag{68}$$

$$t_{o1} = K_{o1} S_o = 552 S_o \tag{69}$$

In the case where memory is searched for contiguous free space greater than the size of the object, if this area is found, the current object will be copied first to the free space and then to the beginning of memory. One should note that if cells are marked every time overflow occurs, then at some point the mark bits have to be reset. If this is not performed, then during the search for a contiguous area in memory, one would also encounter mark bits marked in previous overflows. This may be easily performed within the Copy__Object routine. That is, when an object is copied, whether to the beginning of memory or to a contiguous free space, the mark bits of each copied cell is reset. Therefore, the overhead cost of marking the object and copying it twice is given as:

$$K_{o2}^{0} = 552 + 356 = 908 \tag{70}$$

Searching for a contiguous free area in memory is proportional to the size of memory. Because the routine is not recursive, and since for each cell only the tag is tested and a pointer is incremented if the cell is free, the overall cost per cell is 96 cycles. That is,

$$K_{o2}^{1} = 96 \tag{71}$$

$$t_{o2} = K^0_{o2} S_o + K^1_{o2} M = 908 S_o + 96M \tag{72}$$

The three cases for reallocating an object to the beginning of memory occur with the frequencies $F_1$, $F_2$ and $F_3$ . The average cost of overflow is therefore:

$$K^s_{ov} = F_1 536 S_o + F_2(908 S_o + 96M) + F_3 K_{o3} \tag{73}$$

Whether an object will be reallocated to the beginning of memory directly or in several stages, depends on the object's size relative to the size of memory, that is $\frac{S_o}{M}$. The actual frequencies obtained were computed for each benchmark separately and are shown in the following section.

In the linked list approach, when overflow occurs, the object is marked with the same Mark routine that was used in the sequential approach. The only difference is that maintaining the stack data structure is more expensive here, ( see section 5.2.3.3 ). The rest of the memory is then relinked into a free list. The cost per cell of marking and relinking is given as:

$$K_m = 220 \qquad K_r = 112 \tag{74}$$

Therefore, the average overflow time is:

$$K^l_{ov} = \frac{(N_o + 1)}{N_o} 112M + 220 S_o \tag{75}$$

### 5.5.2.3. Overhead Cost

The implementation overhead is program dependent because it depends on the number of functions executed and the number of cells allocated per primitive function. For the Quicksort benchmark which sorts lists of 8 elements, an average of 2.5 cells were allocated per primitive function. For the Matrix Multiplication benchmark where matrices of size n = 5 were multiplied four times, an average of 1.3 cells were allocated per primitive function. That is,

$$N^{MM}_{apf} = 1.3 \qquad N^{Q}_{apf} = 2.5 \tag{76}$$

The implementation overhead per allocated cell is the same in the sequential and linked list approach. In each case, after executing a primitive function, we have to test whether the Protect register is equal to zero and whether the Maxreached flag has been set. If we first test whether the end of memory was reached and test the Protect register only if Maxreached is set, then every primitive function would have the overhead of 24 cycles.

The functional forms that use the Copy_Object routine would have the extra overhead in incrementing and decrementing the Protect register. This would account for an extra 34 cycles per function. Therefore,

$$K_{pf}^{0} = 24 \qquad K_{pf}^{1} = 34 \tag{77}$$

The frequency with which this extra overhead was encountered in the two benchmarks are:

$$F_{oh}^{MM} = 0.25 \qquad F_{oh}^{Q} = 0.35 \tag{78}$$

The average overhead cost per primitive function and the average overhead per allocated cell for the Matrix Multiplication and Quicksort benchmarks is:

$$K_{pf}^{MM} = 24 + 34 * 0.25 = 32.5 \tag{79}$$

$$\frac{K_{ohv}^{l}}{N_{apf}} = \frac{K_{ohv}^{s}}{N_{apf}} = \frac{32.5}{1.3} = 25 \tag{80}$$

$$K_{pf}^{Q} = 24 + 34 * 0.35 = 36 \tag{81}$$

$$\frac{K_{ohv}^{l}}{N_{apf}} = \frac{K_{ohv}^{s}}{N_{apf}} = \frac{36}{2.5} = 14.4 \tag{82}$$

In the stack memory management policy, the overhead was associated with each primitive function. In some cases, the overhead consists of a PUSH instruction, which, on the Motorola 68000 takes 12 cycles. Some primitive functions do not produce any overhead and some have an extra instruction. In tables 1a and 1d the dynamic frequencies of the executed primitive functions are shown for the Matrix Multiplication and the Quicksort benchmark. One can note that the functions like the Distribute

functions or Split, Div and ID, do not produce any garbage cell. In the Quicksort benchmark, these functions accounted for 26% of all functions used. The Select functions, which accounted for 54% of all functions require only one extra instruction besides a Push instruction. The average overhead cost per instruction for the Quicksort benchmark is found to be 15.12 cycles.

For the Matrix Multiplication benchmarks most of the functions were binary operations and the overhead found is equivalent to a single Push instruction, that is 12 cycles. Therefore one can write:

$$K_{ohv}^{st,MM} = 12 \qquad K_{ohv}^{st,QQ} = 15.12 \tag{83}$$

### 5.5.2.4. Matrix Multiplication Performance Comparison

For the Matrix Multiplication benchmark, the three memory management policies would have the following performances on the Motorola 68000.

$$\frac{T_{seq}}{M} = \frac{N_a^l}{M}(40 + 25) + \tag{84}$$

$$\left[\frac{N_a^l}{(M - S_o)}\right]((552F_1S_o + F_2(908S_o + 96M) + F_3K_{o3}) + 40N_{acv})\frac{1}{M} \tag{85}$$

$$\frac{T_{list}}{M} = \frac{N_a^l}{M}(34 + 25) + 112(\left[\frac{N_a^l}{M}\right] + 1) + 220\frac{S_o}{M}\left[\frac{N_a^l}{M}\right] \tag{86}$$

$$= 59\frac{N_a^l}{M} + (220\frac{S_o}{M} + 112)\left[\frac{N_a^l}{M}\right] + 112 \tag{87}$$

$$\frac{T_{stack}}{M} = 70.5\frac{N_a^l}{M} + \frac{12}{1.3}\frac{N_a^l}{M} = 79.7\frac{N_a^l}{M} \tag{88}$$

In the sequential memory allocation policy, the frequencies $F_1$, $F_2$, $F_3$ , the number of extra allocations due to object reallocation $N_{ac}$, and the average overflow cost $K_{ov}^s$

are computed as a function of the average size $S_o$ of the reallocated object, relative to the size of memory M. This is shown in table 2. The average overflow cost is computed as:

$$K_{ov}^{s} = 552F_1S_o + F_2(908S_o + 96M) + F_3K_3 \tag{89}$$

and the extra number of allocations as:

$$N_{acv} = F_1S_o + 2F_2S_o + nF_3S_o \tag{90}$$

| | $\frac{S_o}{M}$ | | | | |
|---|---|---|---|---|---|
| | 2% | 4% | 6% | 8% | 16% |
| $F_1$ | 97% | 95% | 92% | 90% | 85% |
| $F_2$ | 3% | 5% | 8% | 10% | 15% |
| $F_3$ | 0% | 0% | 0% | 0% | 0% |
| Kov | 559So+3M | 566So+5M | 573So+8M | 580So+10M | 609So+14M |
| Nacv | $1.03S_o$ | $1.05S_o$ | $1.08S_o$ | $1.1S_o$ | $1.15S_o$ |

Table 2. Performance Parameters for the Matrix Multiplication Benchmark

So far, the number of overflows was expressed using the floor function of the ratio of the allocated cells $N_a$ and the size of memory M. If we look at the performance equations only at intervals where overflow occurs, then we can omit the floor function since the ratio will always be an integer value. The three performance equations may then be further simplified to be linearly proportional to $N_a$ / M . They differ only in the constant of proportionality.

$$\frac{T_{seq}}{M} = 65\frac{N_a^l}{M} + \frac{N_a^l}{(M - S_o)}\left(\frac{K_{ov}^s}{M} + 40\frac{N_{acv}}{M}\right) \tag{91}$$

$$= 65\frac{N_a^l}{M} + \frac{N_a^l}{M}\frac{M}{(M - S_o)}\left(\frac{K_{ov}^s}{M} + 40\frac{N_{acv}}{M}\right) \tag{92}$$

$$= \frac{N_a^l}{M}(65 + \frac{1}{(1 - \frac{S_o}{M})}(\frac{K_{ov}^s}{M} + 40\frac{N_{acv}}{M})) \quad (93)$$

$$\frac{T_{list}}{M} = (171 + 220\frac{S_o}{M})\frac{N_a^l}{M} + 1 \quad (94)$$

$$\frac{T_{st}}{M} = 79.7\frac{N_a^l}{M} \quad (95)$$

The time performances of each implementation are compared in table 3. The ratio of the size of the object copied and the size of memory is a parameter.

| | $\frac{S_o}{M}$ | | | | |
|---|---|---|---|---|---|
| | 2% | 4% | 6% | 8% | 16% |
| $\frac{T_s}{M}$ | 80.3 | 95.5 | 113 | 128.5 | 210 |
| $\frac{T_l}{M}$ | 174.68 | 178.36 | 182.04 | 189.4 | 200.44 |
| $\frac{T_{st}}{M}$ | 79.7 | 79.7 | 79.7 | 79.7 | 79.7 |

Table 3. Performance Comparisons for the Matrix Multiplication Benchmark

## 5.5.2.5. Quicksort Performance Comparison

For the Quicksort benchmark, the simplified performance equations are:

$$\frac{T_{seq}^l}{M} = \frac{N_a^l}{M}(58 + \frac{1}{(1 - \frac{S_o}{M})}(\frac{K_{ov}^s}{M} + 40\frac{N_{acv}}{M})) \quad (96)$$

$$\frac{T_l}{M} = (160 + 220\frac{S_o}{M})\frac{N_a'}{M} + 1 \tag{97}$$

$$\frac{T_{sl}}{M} = (71.6 + \frac{15.12}{2.5})\frac{N_a}{M} = 77.6\frac{N_a}{M} \tag{98}$$

The overflow cost and the number of extra allocations are given in table 4.

| | $\frac{S_o}{M}$ | | | | |
|---|---|---|---|---|---|
| | 2% | 4% | 6% | 8% | 16% |
| $F_1$ | 96% | 93% | 90% | 86% | 80% |
| $F_2$ | 4% | 7% | 10% | 14% | 20% |
| $F_3$ | 0% | 0% | 0% | 0% | 0% |
| Kov | 566So+4M | 577So+7M | 588So+10M | 602So+13M | 623So+19M |
| Nacv | $1.03S_o$ | $1.07S_o$ | $1.10S_o$ | $1.14S_o$ | $1.20S_o$ |

Table 4. Performance Parameters for the Quicksort Benchmark

The performance comparisons of the three memory management policies is shown in table 5.

| | $\frac{S_o}{M}$ | | | | |
|---|---|---|---|---|---|
| *Perform.* | 2% | 4% | 6% | 8% | 16% |
| $\frac{T_s}{M}$ | 74.44 | 90.89 | 108.85 | 134.13 | 209.58 |
| $\frac{T_l}{M}$ | 164.68 | 168.36 | 171.04 | 175.72 | 194.44 |
| $\frac{T_{sl}}{M}$ | 77.6 | 77.6 | 77.6 | 77.6 | 77.6 |

Table 5. Performance Comparisons for the Quicksort Benchmark

## 5.6. Performance Discussion

From the performance measurements obtained, we can note that the Linked List memory management policy has the fastest cell allocation time, that is, it is less costly to follow a pointer to a free cell than it is to increment a cell pointer. Another feature of the linked list approach is that it avoids copying of objects, once end of memory is reached. On the other hand, initializing and relinking memory with every overflow creates a significant overhead proportional to the size of memory.

In the case of the Sequential memory management approach, cell allocation cost was slightly higher than the linked list allocation cost. Also, copying of objects to the beginning of memory, presented a significant overhead avoided in the linked list and stack approach. A problem in the sequential approach to memory allocation is that copying objects to the beginning of memory may not always be simple. The frequencies with which the three cases of object reallocation occur are program and data dependent.

If the average size of the copied object is small compared to the size of memory, the overhead of copying is less than the overhead of memory initialization and relinking found in the linked list approach. For these cases the sequential memory management policy will show a better performance. As the average size of the copied object increases, so does the overhead. We can see from the performance tables that the performance of the linked list approach will equal the sequential one if the average size of the reallocated object is approximately 14% of the size of memory. This was the case in both the Matrix Multiplication and in the Quicksort benchmarks.

Therefore, for programs that lead to garbage collection, if the average size of the reallocated object to the beginning of memory is less than 14%, the sequential memory management policy is better than the linked list approach.

A disadvantage of both the sequential and linked list algorithm is that they use the Scratchpad memory extension. Determining the optimal size of this area is again

program and data dependent. In any case, examples can be found so that even scratchpads of large enough size would not be able to bring the execution to a point where objects may be reallocated to the beginning of memory. A mechanism for solving this would have to be implemented.

The stack memory management approach is the only dynamic algorithm considered here. Allocating a cell consists of taking a new cell pointer from the top of the stack and pushing pointers to the remaining garbage data structures back onto the stack. The dynamic garbage collection is therefore part of the memory allocation routine and that is why it is the costliest of the three memory allocation algorithms.

From the performance tables, we can note that the stack approach has a constant time performance for a given class of programs. The sequential approach to memory management outperforms it if the average size of the reallocated object is less than 1% of memory for the Matrix Multiplication benchmark and 2% for Quicksort. One should note, though, that the used benchmarks never lead to the situation which would require the object to be reallocated to the beginning of memory in several stages. If the size of the copied object is kept small, the probability of this happening is quite small.

One should note that even though the stack algorithm maintains a dynamic garbage collection, it is in fact simpler than both the sequential or linked list approach. It does not require a Scratchpad memory extension or a Protect register. It avoids a deadlock situation that may occur in the other two implementations if the scratchpad is not large enough. It also avoids both memory initialization and relinking found in the linked list policy, and object copying found in the sequential approach. It has very little implementation overhead. That is, the only overhead found is included as part of the FP function implementation and cell allocation. It has no overflow overhead since an overflow can not occur if there is at least one available cell in memory.

Therefore, even though the stack approach to cell allocation and immediate garbage collection lead to a significantly higher cost per cell, the overall performance is comparable to the sequential approach for reallocations of objects that are small in size. If the size increases, the stack memory management approach shows a much better performance. The stack algorithm is simpler than both the linked list or stack approach and avoids many of their pitfalls.

### 5.6.1. A Register File Stack Implementation

Until now, the implementation of the stack used in the Stack Memory Management approach was not considered. If we look at figures 17 and 18, one can see that for the MM benchamrk a stack of size 8 is sufficient to manage the garbage collection of a program that required 2752 cells. For the Quicksort benchmark a stack of size 18 is necessary and yet a stack of size 13 would be sufficient to manage over 94% of the allocated cells.

Because of the efficient use of a small number of stack locations, we suggest that the stack of fixed size be implemented in hardware. In fact, the implementation of the hardware stack would be equivalent to a fixed size register file implementation used for garbage collection. If the register file overflows, portions may be moved to memory and brought back into the hardware register file if an underflow occurs.

Using VLSI technology, the register file for garbage collection may be implemented on the processor chip to provide fast cell allocation and efficient dynamic garbage collection.

### 5.6.2. Fast Cell Allocation

Both the sequential and the linked list approach to cell allocation in one way or another, compute the location of the next available cell. We can note that in the stack approach, the cost of allocation is not in computing the address of the available cell, since it is already stored at the top of the stack. Rather, the overhead is in maintaining the garbage data structure, that is pushing onto the stack the appropriate pointers of the allocated cell.

Let us consider a special purpose architecture for the support of each of the three memory management algorithms. In the sequential policy, one could precompute the location of the next available cell by incrementing the current cell pointer and storing the value in a register. In this case, no time penalty would be paid for cell allocation. In the linked list approach, one could follow a pointer to the next free block in memory while the processor is not accessing memory and so implement a 'zero time' cell allocation routine.

In the case of the stack memory management algorithm, the top of the stack contains the next cell pointer. It could be allocated immediately even without hardware support. Nevertheless, it is the storing of the garbage pointers that needs to be

supported by the special purpose architecture. This could be done by previously moving the cell, pointed to by the top of the stack, to two special purpose registers. Once this cell is moved to the processors data section, testing whether the cell is a list or an atom cell, whether any of the pointers are Null pointers and pushing the appropriate pointers onto the stack may be done efficiently and without a time penalty.

Let us consider the performance measurements obtained for the Matrix Multiplication benchmark, but now eliminating the memory allocation cost in all three memory management implementations. The results are shown in table 6.

| | $\frac{S_o}{M}$ | | | | |
|---|---|---|---|---|---|
| | 2% | 4% | 6% | 8% | 16% |
| $\frac{T_s}{M}$ | 40.09 | 55.30 | 73.03 | 90.57 | 165.28 |
| $\frac{T_l}{M}$ | 140.68 | 144.36 | 148.04 | 155.4 | 166.44 |
| $\frac{T_{st}}{M}$ | 12 | 12 | 12 | 12 | 12 |

Table 6. Performance Comparisons for the MM Benchmark Without Allocation Cost

From the above table one can note that by eliminating the allocation time overhead in all three implementations, the stack algorithm is left with only the overhead of placing several extra instructions in each FP function that produces garbage cells. Both the linked list and the sequential approach are left with a memory overflow overhead which has a significant impact on their performance. Because the memory allocation overhead was such a large part of the overall stack memory management overhead, eliminating it would make the performance of the stack algorithm superior to the other two algorithms.

### 5.6.3. An FP Cache

An important property of the stack algorithm, and a direct consequence of the dynamic approach to garbage collection, is that there is a very high locality of object representations in memory. It was shown earlier that the number of used memory

cells at any moment is equal to the size of the current object in memory. This means that the upper bound of used memory will be equal to the size of the largest object created during program execution. In both the sequential and the linked list case, objects 'migrated' during program execution leading to more diverse memory references.

This feature of the stack algorithm leads us to suggest that a cache memory placed between the FP machine and memory may be highly effective. Evaluating the effectiveness of a data cache for FP was beyond the scope of this report, but is part of a continuing research.

### 5.6.4. An FP Multiprogramming environment

If the three memory management policies were to be implemented in a multiprogramming environment sharing a common memory, the sequential approach would have serious difficulties. Memory would have to be divided into different portions for each task or process. Therefore, the overhead would increase linearly, and also the frequency of overflows. One could get into a situation where one task overflows the memory allocated to it, and there is still plenty of memory available, but allocated to other tasks.

The linked list approach avoids the constraint of sequentiality, but has another constraint, that of marking and relinking. If there are several tasks in a common memory, and if the memory allocator reaches the end of memory, all the useful data structures belonging to each taks would have to be marked. Only then may one collect the unused cells into a list of free cells.

With the use of a stack for garbage collection in a multiprogramming environment, all tasks would share the same stack. Since the garbage collection is performed dynamically, and because it is implemented as part fo the FP functions and memory allocation routine, no extra features are necessary. One can note that a multiprogramming implementation with the stack memory management approach is the only implementation that maintains all the properties of a single task implementation. For example, no overflow can occur as long as there is at least one cell that is not used by any of the running tasks. Both the linked list and the sequential approach would lead to an overhead increase proportional to the level of multitasking. In the stack approach the overhead would still be strictly proportional to the number of allocated cells and the number of executed FP functions, therefore transparent to the level of multitasking.

# 6. Conclusion

A uniprocessor implementation of the functional language FP was analyzed in this report. A theoretical model of computation and execution was presented together with implementation constraints.

Two different data structures, Pointer and Sequential, were considered for the representation of the FP objects in memory. Each data structure favored a different class of FP functions but neither showed a strong advantage over the other. A dynamic count of the primitive FP functions is performed on four benchmarks ( Matrix Multiplication (MM), Insertion into a Sorted List (INS), Sieve of Erastothenes (SIEVE) and Quicksort (QUICK) ), in order to gain more insight into the way different primitive functions are used in FP programs. The analysis is not complete so the only observation made is that the most frequent functions are the select primitives. This is intuitive since FP has no variables and the select functions are often used to reference different portions of the list data structures. Memory management constraints were considered for the support of each data structure. The sequential data structure was shown to have serious difficulties in managing the sequential constraint imposed onto the elements of a list structure. The pointer data structure was chosen because of its flexibility and ease of memory management support.

Three memory management policies, Sequential, Linked List and Stack, were considered for the support of the pointer data structure. All three have been simulated and implemented using an already existing FP interpreter. The Stack approach was the only memory management policy considered that performed dynamic garbage collection. Theoretical performance estimates were made for each memory management policy, considering three types of overhead: memory allocation overhead, overflow and implementation overhead.

Cell allocation was fastest in the linked list approach, and slowest in the stack approach. The stack allocation had the worst allocation time per cell because it included garbage collection operations, ( one must have in mind that these results do depend on the host machine, that is, in our case the implementation was on a Motorola 68000 microprocessor.

The linked list approach had a high overflow overhead proportional to the size of memory. The sequential overflow overhead was proportional to the average size of the object copied to the beginning of memory, once an overflow occurred. The stack approach had no overflow overhead since it performed dynamic garbage collection.

The performance estimates were followed by benchmark results implemented on an off_the_shelf microprocessor, the Motorola 68000. Due to the high cost of cell allocation in the stack approach, the sequential policy outperforms it if the average size of the copied object is less than 1-2% of the size of memory. The benchmarks used favored the sequential approach since it never lead to conflict ( copying the object to the beginning of memory in several stages ) or deadlock ( a large enough Scratchpad memory extension was used ) situations. If the average size of the copied object in the sequential approach is greater than 2%, the stack approach is better for the given benchmarks.

The stack approach eliminates the possibility of a conflict or a deadlock situation occurring. It is more amenable to multiprogramming and shows a very high locality of object representation in memory, thus suggesting the possibility of an efficient data cache support.

Because of the unique way in which the garbage data structures are managed in the stack memory management approach, the stack size was kept small. Therefore we suggest a hardware register file implementation of a fixed size stack used only for garbage collection. Using available VLSI technology the register file may be implemented on the processor chip. A mechanism for stack overflow and underflow must be provided.

A special purpose architecture was considered for the support of fast cell allocation. Since cell allocation was the major part of the overhead in the stack implementation, such an architecture would benefit it most, thus leading to an implementation superior to both the sequential and the linked list approach.

# 7. Appendix A

## 7.1. The Matrix Multiplication Benchmark

Def MM ≡ (AP AP IP) @ (AP DistL) @ Distr @ {1,Trans @ 2};

Def IP = ( IN + ) @ ( AP * ) @ Trans;

The Matrix Multiplication FP program described here takes as an input object a list of two matrices each represented as a list of rows. That is, for matrices A and B, the input would look like:

X ≡( ( $r_{a1}$ , ... , $r_n$), ( $r_{b1}$ , ... , $r_{bn}$)

where each row is a list, that is $r_{a,bi}$ = ( $a_{i1}$... $a_{im}$).

## 7.2. Insertion Into A Sorted List Benchmark

Def INSERT ≡ ( > @ { SL2, SL1 @ SL3 } →

INSERT @ {APENDL @ {SL1 @ SL3, SL1},SL2,TAIL @ SL3},

CONC @ { REV @ SL1, { SL2 }, SL3 } );

The Insert FP program operates on a 3 element list, ( $l_1$, $l_2$, $l_3$) where $l_1$ is initially null and $l_2$ is the element to be inserted into the sorted list $l_3$.

## 7.3. The Sieve Of Erastothenese Benchmark

Def SIEVE ≡ RSIEVE @ { K(), SL1, TL };

```
Def RSIEVE = IF ( NL @ SL3, REV @ SL1,
    IF ( NOT @ { EQ, { K0, % { SL1 @ SL3, SL2 } },
    RSIEVE @ { APL { SL1 @ SL3, SL1 }, SL2, TL @ SL3 },
    RSIEVE @ { SL1, SL2, TL @ SL3 };
```

## 7.4. The Quicksort Benchmark

```
Def Quick = IF ( > @ { LN,K1 },
   CT @ " QQ1,VV1 " @ SSORT @ " MID, SP ",ID);
Def QQ1 = QUICK @ qq;
Def VV1 = QUICK @ vv;
Def SSORT = {FIL @ SET @ {SL1 ,SL2 @ SL2},FIL @ SET @
    { SL1, SL1 @ SL2 }};
Def MID = PK @ { / @ { ID, K2 } @ LN, ID };
Def SET = { ID, K(), K() }, DL };
Def vv = CT @ { SL3 @ SL1, SL2 @ SL2 };
Def qq = CT @ { SL2 @ SL1, SL2 @ SL2 };
Def FIL = IF( NL @ SL1, ID,
    FIL @ IF(| @ {EQ @ SL1 @ SL1, GT @ SL1 @ SL1},
    {TL @ SL1, AR @ { SL2, SL2 @ SL1 @ SL1 }, SL3},
    {TL @ SL1, SL2 , AR @ {SL3, SL2 @ SL1 @ SL1}};
```

# 8. Appendix B

## 8.1. FP Memory Simulator Support Routines

The Memory Management Simulator was built using an already existing FP interpreter ( see Chpater 3). Both the interpreter and the memory simulation routines were written in C. Throughout this report pseudo pascal was used to describe the memory allocation and garbage collection algorithms with the intention of making them more readable and easier to understand. In this appendix the additional memory management support routines are presented in their original version, that is in C.

Also, in the memory management algorithms described in Chapter 5, the term "pointer" was used to refer to addresses of memory cells. In the actual simulator, memory was represented as an array of cells, so that indexes were used instead of pointers. This is shown in figure 19.1 together with other data structures that were used for gathering memory management related information.

```
struct cell { int tag;
              int use;
              int next;
              int value;} memory(maxsize);
int stack(stacksize);
int stackfreq(stacksize);
int stackhistory(stacksize/2);
int stackpointer;
int copycount;
int cellcopiedcount;
```

It is not our intention to describe all the memory management support routines that are used in the simulation, they are available on line. Instead, only the routines that were mentioned earlier in Chapter 5 are presented, that is CreateObject, DisplayObject, ContiguousAreaFound and ReallocateIteratively.

The CreateObject routine takes a pointer to a string representing the input object, as a parameter. It returns an interger value which is an index into memory, pointing

to the first cell in the object data structure.

```
int CREATE_OBJECT(str,ptr)
    char str();
    int *ptr;
    {
     int allocatecell();
     int p,q,h,lastp,i,ip;
     char token(2);
     h = Null;
     ip = *ptr;
     while(str(ip) == ' ' ) ip ++;
     while(str(ip) !=')' && str(ip))
      {
        while(str(ip) == ' ') ip++;
        if(str(ip) == ' ') ip++;
        if(str(ip) == '(')
        {
          ip++;
          p = Create_Object(str,&ip);
          q = Allocatecell(&x);
          mem(q).tag = 1;
          mem(q).value = p;
          mem(q).next = Null;
          if (h == Null) h = q;
          else mem(lastp).next = q;
          lastp = q;
          if(str(ip) != ')') printf(%s,"no left paren");
          ip++;
        }
        elseif (isxdigit(str(ip)))
        {
          for(i = 0; isxdigit(str(ip) && str(ip);
                     token(i++) = str(ip++));
          token(i) = '\0';
          num = atoi(token);
          p = allocatecell(&x);
          mem(p).tag = Atom;
          mem(p).next = Null;
          mem(p).value = num;
          if (h == Null) h = p;
          else mem(lastp).next = p;
          lastp = p;
        }
     }
     *ptr = ip;
     return(h);
```

```
   }
```

The DisplayObject routine will display the object pointed to by the index into memory called pointer.

```
Display_Object(pointer)
int pointer;
 {
   int p;
   p = point;
   while(p != Null)
   {
      if ( mem(p).tag )
      {
         printf("%c",')');
         Display_Object(mem(p).value);
         printf("%c",')');
         if ( mem(p).next != Null )
            printf(",");
      }
      else
      {
         printf("%d",mem(ip).value);
         if (mem(ip).next != Null) printf(",");
      }
      p = mem(p).n;
   }
 }
```

Given a cell_count value determined during marking, the ContiguousAreaFound routine will check if an area exists so that the object may be copied there before it is copied to the beginning of memory.

```
int Contiguous_Area_Found(cell_count)
int cell_count;
    {
     int cellp1, cellp2;
     contiguous_area_found = 0;
     while(! contiguous_area_found)
       {
        cellp1 = cellp2;
        while( mem(cellp1).tag == 'USED' )
            cellp1++;
        cellp2 = cellp1;
        while( mem(cellp2).tag != 'USED' )
            cellp2++;
```

```
        if (cellp2 - cellp1 > cell_count)
            contiguous_area_found = 1;
       }
    }
```

The ReallocateIteratively routine will copy the object into several different locations, starting with the largest contiguous area found during the ContiguousAreaFound routine.

```
ReallocateIteratively(maxtarea,location )
int maxarea;
int location;
    {
     while( !contiguous_area_found )
        {
        free_cell_pointer = location;
        CopyObjectI(cell_pointer);
        }
    }
```

One should note that in the ReallocateIteratively routine shown above, the CopyObjectI routine is not the same as the CopyObject routine described earlier in figure 8.2 . There is one modification that has to be made to the allocation routine. Instead of it allocating always by incrementing a pointer, the allocation routine will allocate a cell only if its tag is unmarked. This call to the modified sequential memory management allocation routine is made only during the iterative allocation process.

# 9. Appendix C

The number of periods shown in the following tables includes instruction fetch and all applicable operand fetches and stores. The number of bus read and write cycles is shown in parenthesis as: (r,w). If the number of cycles is followed by a +, then the effective address calculation time must be added. The * symbol following the read and write cycles means that the instruction takes a total of 8 clock periods if the effective address is register direct. In table 7 a subset of the Motorola 68000 instruction set is shown together with the number of clock periods.

| Instruction | Clock Periods |
|---|---|
| move.W r,r | 4 (1,0) |
| move.L r,r | 4 (1,0) |
| move.W r,r@ | 8 (1,1) |
| move.W r@,r | 8 (2,0) |
| move.L r,r@ | 12 (1,2) |
| move.L r@,r | 12 (3,0) |
| add.w (ea),D | 4 (1,0)+ |
| add.w (ea),A | 8 (1,0)+ |
| add.w D,<M> | 8 (1,1)+ |
| add.L (ea),D | 6 (1,0)+* |
| add.L (ea),A | 6 (1,0)+* |
| add.L D,<M> | 12 (1,2)+ |
| and.w ea,D | 4 (1,0)+ |
| and.w D,<M> | 8 (1,1)+ |
| and.L ea,D | 6 (1,0)+* |
| and.L D,<M> | 12 (1,2)+ |

| Instruction | Clock Periods |
|---|---|
| cmp.W ea,A | 6 (1,0)+ |
| cmp.w ea,D | 4 (1,0)+ |
| cmp.L ea,A | 6 (1,0)+ |
| cmp.L ea,D | 6 (1,0)+ |
| bcc.b taken | 10 (2,0) |
| bcc.w taken | 10 (2,0) |
| bcc.b not taken | 8 (1,0) |
| bcc.w not taken | 12 (2,0) |
| bra.b taken | 10 (2,0) |
| bra.w taken | 10 (2,0) |
| bsr.b taken | 18 (2,2) |
| bsr.w taken | 18 (2,2) |
| trap taken | 34 (4,3) |

Table 7. Motorola 68000 Instruction Clock Periods

# Bibliography


[AckDen79] Ackerman,W. and Dennis, J., *VAL - A Value Oriented Algorithmic Language,* MIT LCS TR-218 (1979).

[Alkal84] Alkalaj,L. and Monsalve,L., *Fast FP Implementation On The Motorola 68000,* Class Project In High Speed Computing CS 259, UCLA (1984).

[Arvind78] Arvind and Gostelow,K.P. and Plouffe,W., *An Asynchronous Programming Language and Computing Machine,* Univ. of California Irvine, CA, Tech. Rep. 114,Dec 78 (1978).

[Backus78] Backus, J., *Can Programming Be Liberated From The Von Neumann Style,* CACM VOL.21, No.8 (1978).

[Backus80] Backus, J., *Functional Level Programs as Mathematical Objects,* Proc. Functional Programming Languages and Computer Architecture (1980).

[Castan83] Castan,M. and Organick,E, *An HLL RISC Processor For Parallel Execution Of FP Language Programs,* The 9th Annual Symposium On Computer Architecture (1983).

[Cohen67] Cohen,J. and Trilling,L., *Remarks On Garbage Collection Using a Two Level Storage,* BIT 7, 1 (1967).

[Cohen81] Cohen,J., *Garbage Collection of Linked Data Structures,* ACM Computing Surveys, Vol.13, No.3, September 81 (1981).

[Feller81] Feller,M., *CS 259 Term Project, UCLA,* UCLA CS Dept. (1981).

[Flynn83] Flynn,M. and Hoevel,L., *Execution Architecture: The DELtran experiment,* Transactions on Computers C-32 Feb. (1983).

[HarEva64] Hart,T.P. and Evans,T.G., *Notes On Implementing Lisp For the M460 Computer,* MIT, Cambridge, Mass., (1964).

[Johnst71] Johnston,J.B., *The Contour Model Of Block Structured Processes,* SIGPLAN Notices Feb 1971 (1971).

[Kellm83] Kellman,J., *Parallel Execution of Functional Programs,* UCLA Master's Report (1983).

[Kelle81] Keller,M.R. and Sleep,M.R., *Applicative Caching,* Proc. ACM Conf. Functional Programming Languages and Computer Architecture 1981 (1981).

[Knuth73] Knuth,D., *The Art Of Computer Programming,* Addison-Wesley (1973).

[Lahti80] Lahti,D., *Application Of A Functional Programming Language,* UCLA Master's Thesis (1982).

[Land65] Landin,P.J., *A Correspondence Between Algol60 And Church's Lamda Notation,* Communications ACM Vol 8. Feb 1965 (1965).

[Mago80] Mago,G., *A Cellular Computer Architecture For FP,* COMPCON 80 Digest Of Papers, San Francisco (1980).

[McCa65] McCarthy,J., et al., *LISP 1.5 Programmer's Manual,* MIT Press, Cambridge, Mass., (1965).

[McGr79] McGraw,J.R., *Data Flow Computing: Software Developement,* Proc. IEEE Int. Conf. Distributed Computing Systems (1979).

[Monsal84] Monsalve,L., *Performance Comparison Of Two M68000 Runtime Environments, FP and Pascal,* Class Project In High Speed Computing CS 259, UCLA (1984).

[Morr80] Morris,J. and Schmidt,E. and Wadler,P., *Experience With An Applicative String Processing Language,* Proc. ACM Symp. Princ. Programming Languages July 1980 (1980).

[Motor82] Motorola, *MC 68000 16-bit Microprocessor User's Manual,* Prentice Hall, Inc., Englewood Cliffs, N.J. (1982).

[Patel80] Patel, D., *A System Organization For Applicative Programming,* UCLA Master's Thesis (1980).

[ShihLi84] Shih-Lien,L., *A Compiler For Functional Programming System,* UCLA Master's Thesis (1984).

[Schor67] Schorr,H. and Waite,W., *An Efficient Machine-Independent Procedure for Garbage Collection In Various List Structures,* Communications ACM 10, 8 Aug. 1967 (1967).

[Turner82] Darlington, Henderson and Turner, *Recursive Equations as a Programming Language,* Cambridge University Press (1982).

[Turner81] Turner,D.A., *The Semantic Elegance of Applicative Languages,* Proc. Functional Programming Languages and Computer Architecture (1981).

[TrMo80] Treleaven,P. and Mole,G., *A Multiprocessor Reduction Machine for User Defined Reduction Languages,* University of Newcastle upon Tyne, Technical Report number 150 (1980).

[Vegd84] Vegdahl, S., *A Survey Of Proposed Architectures For The Execution Of Functional Languages,* IEEE Transactions on Computers, Vol C-33 No.12 Dec 84 (1984).

[Xiong84] Xiong, *An FP Machine Based On Queues,* UCLA Master's Thesis (1980).